Dynamic Data Structures and Saliency-influenced Rendering

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Daniel Schiffner

aus Aschaffenburg am Main

Frankfurt 2012

(D 30)

vom Fachbereich Informatik und Mathematik (12) der

Johann Wolfang Goethe-Universität als Dissertation angenommen.

Dekan: Prof. Dr. Tobias Weth

Gutachter: Prof. Dr. Detlef Krömker

Prof. Dr. Ralf Dörner

Prof. Dr. Ulrich Schwanecke

Datum der Disputation:

Zusammenfassung

Durch die Entwicklung moderner und heterogener Hardware haben sich die Anforderungen an 3d Anwendungen geändert. Obwohl Echtzeit-Rendering von photorealistischen Bildern heutzutage möglich ist, bleibt der Rechenaufwand trotz leistungsfähiger Graphikkarten enorm hoch. Gerade ältere Computer mit schwächerer Graphikkarte oder Smart-Phones sind nicht in der Lage die selben Ergebnisse zu erreichen. Um auf diesen leistungsschwächeren Systemen trotzdem ein interaktives Rendering zu ermöglichen, werden üblicherweise Details von den Objekten innerhalb einer Szene entfernt. Dadurch müssen weniger Daten verarbeitet werden und als Resultat wird eine höhere Bildwiederholfrequenz erzielt. Das Entfernen von Details kann dabei jedoch Artefakte erzeugen, welche von einem Betrachter oder Benutzer als störend empfunden werden können. Dadurch wird die vom Betrachter empfundene Qualität der Szenerie reduziert.

Um die Erzeugung von Artefakten zu vermeiden, werden Merkmale auf einem Objekt identifiziert, welche entfernt werden können. Meistens werden dazu geometrische Eigenschaften genutzt, wie z.B. Abstand, Fläche oder Form. Mit Hilfe von diesen wird die Qualtität einer möglichen Reduktion eingeschätzt. Die wiederholte Anwendung von diesen Reduktionen erzeugt die sogenannten Levels Of Detail (LODs), welche von einem Renderingsystem genutzt werden können. Diese vorberechneten LODs werden dann mittels einer Metrik, wie der Abstand zum Betrachter, ausgewählt und ausgetauscht. Der Übergang zwischen zwei LODs-Versionen muss dabei kontinuierlich von statten gehen, um von einem Betrachter nicht wahrgenommen zu werden. Daher ist es nötig, beide LOD-Versionen gleichzeitig anzuzeigen und zwischen beiden überzublenden, d.h. mit Hilfe der Transzparenz einen Austausch der Objekte zu vollführen. Der dadurch entstehende zusätzliche Aufwand muss hierbei in Betracht gezogen werden und der Vorteil einer LOD-Reduktion ist erst nach dem Vollenden des Austausches gegeben. Grundsätzlich nutzen die aktuellen LOD-Methoden die diskreten und vorberechneten Level. Dabei werden aber auch Beschränkungen von Anzeige und Betrachter ausgenutzt.

Die visuelle Wahrnehmung von Menschen ist limitiert, z.B. können wir uns nur auf eine Stelle gleichzeitig fokussieren oder gleiche Farben nicht unterscheiden, wenn die Lichtverhältnisse verschieden sind. In der Forschung innerhalb dieses Bereiches wurden verschiedene Kompressionsarten vorgestellt, welche diese Informationen nutzen können. Bildbasierte Kompressionen, wie MPEG und JPEG, nutzen diese, um für Menschen unbemerkbare Details zu entfernen. Die JPEG-Bildkomprimierung zum Beispiel macht sich unter Anderem zu Nutze, dass Menschen eine geringere Auflösung für Farbeninformationen besitzen. Auch in anderen Wahrnehmungsbereichen finden Kompressionsmethoden Anwendung, wie z.B. MP3. In dieser Kompression werden überlagerte Frequenzen mit geringerer Genauigkeit speichert, falls diese von einer anderen Frequenz maskiert wird. Zur Abbildung von Wahrnehmung, insbesondere die visuelle Wahrnehmung, gibt es verschiedene Computermodelle. In unserem Szenario sind besonders solche von Vorteil, welche nicht von Menschen beeinflusst werden

können, wie z.B. die Salienz.

Salienz ist ein Begriff aus der Psychophysik und beschreibt die Fähigkeit eines Objektes aus seiner Umgebung “hervorzustechen”. Solche Objekte (oder Merkmale) sind besonders wichtig für das menschliche visuelle System. Mit Hilfe der Salienz ist es möglich, Regionen zu identifizieren, auf die ein Mensch wahrscheinlich schauen wird. In der Literatur wurde die Salienz bereits verwendet, um rekursive Rendering-Algorithmen zu steuern, wie z.B. Path-Tracing oder globale Beleuchtungsrechnungen. Gerade sehr aufwändige Verfahren ziehen einen großen Nutzen aus einer wahrnehmungsbasierten Darstellung, da sie dadurch einzelne Berechnungen abbrechen können ohne die Qualität des Ergebnisbildes stark zu beeinflussen. Im Idealfall würde das Ergebnisbild sich nicht von einer naïven Methode unterscheiden. Bis jetzt ist es allerdings üblich, Salienz für Bilder zu berechnen und eine 3d Variante wurde nur partiell vergestellt und umgesetzt. Damit das existierende Modell komplett in 3d überführt werden kann, müssen einige Probleme umgangen werden.

In dieser Arbeit stellen wir ein smartes Renderingsystem vor, welches mit Hilfe eines 3d Salienzmodells Wahrnehmungsinformationen nutzen kann. Dieses Rendering passt die vorhandenen Objekte direkt während der Anzeige an und ermittelt mit der Salienz Bereiche oder Regionen, in denen Details entfernt werden können. Dadurch ist dieses System nicht auf Vorberechnungen bezüglich der LOD-Versionen angewiesen. Wir präsentieren eine allgemeine Repräsentation der Salienz, welche es ermöglicht, einzelne Merkmale im Voraus zuberechnen und in einer beliebigen Szenerie einzufügen. Unser LOD-Verfahren ist dabei nicht nur abhängig von Objektinformationen, sondern nutzt zusätzlich die Szenerie, wodurch genauere Anpassungen möglich sind. Zur Bestätigung unserer Thesen, haben wir einen Prototypen erstellt, der auf aktuellen Systemen interaktives Rendering ermöglicht und konnten in Benutzerstudien die vorgestellten Verfahren verifizieren.

Für die Anpassung eines 3d Objektes definieren wir eine dynamische Datenstruktur, den *TreeCut*. Diese kapselt eine multi-resolution Darstellung ab, welche verschiedene Detailstufen eines Objektes enthält. Im Gegensatz zu klassischen, hierarchiebasierten Renderingmethoden, wird in unserem Verfahren explizit ein Cut gespeichert, welcher die aktuelle Darstellung repräsentiert. Diese Darstellung wird dabei durch die sogennanten Cut-Knoten beschrieben. Es ist nicht nötig, die hierarchie nochmals zu traversieren und eine Beschleunigung in der Anzeige wird erreicht, da die Darstellung allein durch die Cut-Knoten möglich ist. Zusätzlich kann, auf Grund dieser expliziten Darstellung, der *TreeCut* mit nur zwei Basisoperationen verändert werden, das `refine` und `coarse`. Wird die `refine`-Operation angewendet, so erhöht sich das Detail an der aktuellen Stelle, wohingegen die `coarse`-Operation dieses dort entfernt. Beide Operationen sind lokal, d.h. es werden keine globalen oder externen Informationen benötigt. Lediglich Nachfolger- oder Vorgängerinformationen müssen zugänglich sein, um einen Austausch zu ermöglichen.

Wir stellen verschiedene Evaluationsstrategien vor, um eine *TreeCut*-Repräsentation nicht nur lokal sondern global zu verändern. Dafür werden einem Cut-Knoten entweder Prioritäten oder ein Ziellevel (Bucket) zugewiesen. Im Fall der Prioritäten kann ein Optimierungsalgorithmus die Gesamtpriorität eines *TreeCut*s maximieren. Dafür ist eine Beschränkung nötig, wie zum Beispiel die maximale

Anzahl von Primitiven oder Cut-Knoten. Die Evaluationsstrategie ersetzt dann mittels der beiden Basisoperationen Cut-Knoten mit geringer Priorität durch andere mit höherer Priorität. Im Falle des Buckets wird der *TreeCut* solange verändert, bis die vorgegebene Verteilung erreicht wurde. Beide Evaluationsstrategien arbeiten mit linearer Komplexität in Abhängigkeit von der Größe des *TreeCut*s.

Um eine effiziente Auswertung und Darstellung zu erreichen, werden Rendering und Evaluation parallel verarbeitet. Dabei wird das Objekt über mehrere Frames angepasst, ohne das Rendering zu blockieren. Wir stellen innerhalb der Arbeit eine Parallelisierungsmethode vor, um den Austausch zwischen der Evaluationsstrategie und der Darstellung zu ermöglichen. Dafür wird die hierarchie Information von der Darstellung getrennt. Trotz der zusätzlichen Berechnungen und Trennung der Daten behält der *TreeCut* seine lineare Zeitkomplexität für das Rendering und die Auswertung.

Der *TreeCut* ist in seiner Anpassungsfähigkeit nicht auf geometrische Eigenschaften beschränkt und als eine mögliche Alternative wird eine Stippling-Methode präsentiert. Ein Stippling ist eine punkt-basierte Darstellung eines Objektes mit verschiedener Punktdichte, um die Silhouette oder interne Merkmale darzustellen. In diesem Fall wird die Bucket-Strategie genutzt, um mittels der Beleuchtung die korrekte Punktdichte zu ermitteln. Die multi-resolution Darstellung wird in diesem Fall angepasst, um verschiedene Punktdichten zu liefern. Der *TreeCut* kann auch dazu genutzt, eine bestimmte Wichtigkeit eines Objektes sicherzustellen oder zu erzeugen. In einem Testszenario haben wir ein Icon so angepasst, dass es sich in Abhängigkeit von Benutzer und Aufgabe verändert. Damit konnten wir einen Zusammenhang zwischen Benutzerbelastung und der Priorität des Icons zeigen. Werden diese Informationen zusammengetragen, wäre es möglich ein Benachrichtigungssystem zu erstellen, welches den aktuellen Zustand des Benutzers berücksichtigen kann.

Optimiert man eine Szenerie oder ein Objekt können wahrnehmungsorienterte Methoden die Beschränkungen des visuellen Systems berücksichtigen. Existierende Salienzverfahren erzeugen dazu ein 2d Bild, welches Bereiche identifiziert, in denen Detail erhalten werden sollte. Diese Salienz ist dabei abhängig von mehrerern Faktoren, unter Anderem die Sicht und Beleuchtung der Szenerie. Wir präsentieren ein 3d Salienzmodell, welches eine universelle Berechnung von Wahrnehmungsinformationen erlaubt, die Bidirectional Saliency Weight Distribution Function (BSWDF). Zur Bestimmung der Merkmale werden direkt 3d Informationen genutzt, um eine Approximation aus 2d Daten zu vermeiden. Dazu werden vorhandene Merkmalsextraktoren (engl. feature extractors) von 2d in den 3d Raum übertragen. Durch die allgemeine Beschreibung der BSWDF ist es damit auch möglich, Bilder zu bearbeiten und in diesen wichtige Regionen zu identifizieren.

Um die einzelnen Merkmale zu extrahieren, benutzen wir sowohl Fähigkeiten von modernen Graphikkarten als auch aktuelle Verfahren aus der Computergraphik. Inspiriert von punkt-basierten Renderingmethoden werden die einzelnen Merkmale in einem einzelnen Element (dem sogennanten surface element (surfel)) gespeichert und auf Salienz untersucht, d.h. wie sehr es “hervorsticht”. Dabei beschränkt sich die vorgestellte Methode nicht nur auf punkt-basierte Repräsentationen, sondern kann für alle Primitivtypen benutzt werden. Zur Auswertrung verwenden wir ein Shader-Programm, welches auf der Graphics Processing Unit (GPU) ausgeführt wird und dadurch eine Übertragung von

Daten vermeidet. Zusätzlich wird die Berechnung, wegen der schnellen Graphikkarte, beschleunigt. Die so erhaltenen objektbasierten Merkmale können extrahiert und zusammengefasst werden, um eine Salienzkarte für die Objektoberfläche zu erstellen.

Diese oberflächenspezifische Merkmale, wie Farbe oder Krümmung, können dabei vorberechnet werden, um Rechenzeit während des Renderings zu sparen. Wir stellen hierfür Sampling Schemata vor, um die einzelnen Blickpositionen zu bestimmen, die däfur benötigt werden. Mit diesen ist sichergestellt, dass während des Renderings in jedem Blickwinkel alle Oberflächenmerkmale rekonstruiert werden können. Die Ergebnisse aus den Sampling Schemata werden innerhalb einer Lookup Table gespeichert, ähnlich zu Verfahren, wie sie in der Beleuchtungsrechnung Anwendung finden. Die Größe der Tabelle ist dabei nur von Anzahl der Blickpositionen und der genutzten Bildgröße abhängig. Die geometrische Komplexität, d.h. die Anzahl der Primitive, welche das Objekt beschreiben, beeinflusst die Lookup Table nicht. Dies bedeutet, dass die Qualität einer Lookup Table unabhängig von einem Objekt ist.

Die Berechnung der BSWDF kann sowohl auf der Central Processing Unit (CPU) oder der GPU erfolgen und nur wenige Instruktionen werden dafür benötigt. Für den Fall, dass die Daten vorberechnet wurden, wird eine Rückprojektion durchgeführt und mit den aktuellen wahrnehmungsrelevanten Informationen aus der Szenerie verknüpft. Zusammen mit einem speziellen Beleuchtungsmodell wird die Salienz für alle Primitive berechnet. Falls die Daten auf der GPU berechnet werden, werden diese mit Hilfe der “transform feedback” Fähigkeiten übertragen. Im Gegensatz zu anderen Verfahren, erhält das “transform feedback” dabei die verwendete Ordnung der Primitive und erlaubt dadurch eine schnelle Zuweisung der ermittelten Salienzdaten. Interpretiert man diese Salienzdaten als Prioritäten, kann zusammen mit den *TreeCut* Evaluationsstrategien ein Objekt wahrnehmungsorientiert angepasst werden.

Eine so durchgeführte Anpassung eines Objektes basiert dabei auf Szenen gestützten Informationen und eine Änderung resultiert somit in neuen Salienzinformationen. Ein Objekt ist durch den *TreeCut* und der BSWDF in der Lage, sich optimal der aktuellen Szenerie anzupassen. Das Objekt wird somit selbst-optimierend und ein *Feedback System* erschaffen. Das Ergebnis der wiederholten Anpassung ist wahrnehmungsoptimiert. Um die Nützlichkeit von Salienz zu bewerten, wurden mehrere Benutzerstudien durchgeführt, welche die Qualität der erzielten Reduktionen bewerteten. Wir haben in diesen Studien eine Salienzkompression mit einer geometrischer Kompression verglichen. Ein Resultat ist unter Anderem, dass die Salienzinformationen es erlauben ein Objekt noch weiter zu komprimieren als es mit rein geometrischen Verfahren möglich wäre. Dabei wird die Qualität der Reduktion nicht beeinflusst obwohl die Anzahl der Primitive verringert wurde. Die Teilnehmer waren zum Beispiel nicht in der Lage zwischen zwei Objekten zu unterscheiden, obwohl die Salienzkomprimierung das Objekt nur mit 60% der Größe der geometrisch-reduzierten Variante darstellte. War dieses Verhältnis größer, so wurde die Salienzvariante bevorzugt. Mit Hilfe von statistischer Analyse konnten wir die Ergebnisse als signifikant bzw. als hoch-signifikant einstufen.

Das *Feedback System* erlaubt es, ein beliebiges Objekt in seiner Darstellung zu optimieren. Aber

nicht nur geometrisches Detail kann damit geändert werden und als Beispiel stellen wir eine dynamische Simulation vor. Diese basiert auf einem Software Development Kit (SDK), um sowohl die universelle Einsetzbarkeit des *Feedback System*s zu zeigen und den Fokus auf die Umsetzung der Verknüpfung zu legen. In diesem Szenario dient der *TreeCut* dazu, nicht nur geometrisches- sondern auch simulations-bezogenes Detail zu verändern. Wir fügen das SDK in das vorhandene System ein und passen dazu den *TreeCut* entsprechend an. Durch die Anpassung des *TreeCut*s wird zusätzlich zur Beschleungigung der Darstellung, auch die Rechenzeit für einen Simulationsschritt reduziert.

In der Simulation verwenden wir einen sogennanten Soft-Body, welches ein deformierbarer aber auf der Oberfläche verbundener Körper ist. Dies trifft zum Beispiel auf ein Stück Stoff zu, welches sich an eine Oberfläche anpassen kann, aber durch das Geflecht nicht auseinander reißt. Andere Typen sind starre Körper – dies sind idealisierte Körper, welche nicht elastisch sind – und Fluide oder Gase. Die beiden Letzten werden vor allem in punkt-basierten Simulationen verwendet. Jede dieser Simulationsarten ist dabei von der Anzahl der zu berechnenden Simulationsknoten abhängig und eine Reduktion liefert entsprechend eine große Beschleunigung. Wir definieren für dieses Szenario eine besondere BSWDF, welche animationsbasierte Informationen nutzt, wie z.B. die Bewegung eines Objektes oder eines Knotens. Das *Feedback System* kann mit Hilfe dieser BSWDF den Detailgrad in sich schnell bewegenden Regionen erhöhen und in statischen Bereichen Detail entfernen. Die Operationen berücksichtigen dabei auch die Wahrnehmung von Menschen und erhalten bei einer Reduktion wichtige Informationen.

Die Verwendung von Wahrnehmung im Echtzeitrendering ist ein wichtiges Thema aktueller Forschung. Das menschliche visuelle System ist inzwischen hinreichend erforscht und gültige Computermodelle wurden entwickelt. Diese finden sowohl in kommerziellen als auch in freien Programmen anwendung, wie z.B. im Falle der JPEG-Komprimierung. In dieser Dissertation wird der *TreeCut* vorgestellt, welcher das LOD eines Objektes, unabhängig von seiner Repräsentationsart, dynamisch und kontinuierlich anpassen kann. Spezifische Level müssen hierbei nicht bereitgestellt oder vorberechnet werden. Zusammen mit der Identifikation von wichtigen Regionen durch die BSWDF, wird eine wahrnehmungsbasierte Auswertung eines Objektes ermöglicht. Im Gegensatz zu bisherigen Salienzmodellen, werden innerhalb der BSWDF explizit 3d Daten genutzt. Ein Teil der zu berechnenden Informationen kann ausgelagert werden, um die dafür benötigte Rechenzeit während des Renderings und während der Berechnung der BSWDF zu sparen. Diese oberflächenspezifischen Informationen können innerhalb eines beliebigen Beleuchtungsszenarios eingefügt werden Das *Feedback System*, bestehend aus dem *TreeCut* und der BSWDF, optimiert dabei die Darstellung. Diese Optimierung ist nicht nur auf visuelle Eigenschaften beschränkt. Wir konnten mit einem Prototypen interaktives Rendering erzielen und haben die Ergebnisse des *Feedback System*s mit Hilfe von Benutzerstudien validiert. Jedoch beschränkt sich das vorgeschlagene System nur auf spezifische Aspekte des wahrnehmungsbasierten Renderings und weitere Forschung in dem Bereich ist nötig, um noch mehr Nutzen aus den Wahrnehmunginformationen ziehen zu können.

Abstract

With increasing heterogeneity of modern hardware, different requirements for 3d applications arise. Despite the fact that real-time rendering of photo-realistic images is possible using today's graphics cards, still large computational effort is required. Furthermore, smart-phones or computers with older, less powerful graphics cards may not be able to reproduce these results. To retain interactive rendering, usually the detail of a scene is reduced, and so less data needs to be processed. This removal of data, however, may introduce errors, so called artifacts. These artifacts may be distracting for a human spectator when gazing at the display. Thus, the visual quality of the presented scene is reduced.

This is counteracted by identifying features of an object that can be removed without introducing artifacts. Most methods utilize geometrical properties, such as distance or shape, to rate the quality of the performed reduction. This information used to generate so called Levels Of Detail (LODs), which are made available to the rendering system. This reduces the detail of an object using the pre-calculated LODs, e.g. when it is moved into the back of the scene. The appropriate LOD is selected using a metric, and it is replaced with the current displayed version. This exchange must be made smoothly, requiring both LOD-versions to be drawn simultaneously during a transition. Otherwise, this exchange will introduce discontinuities, which are easily discovered by a human spectator. After completion of the transition, only the newly introduced LOD-version is drawn and the previous overhead removed. These LOD-methods usually operate with discrete levels and exploit limitations of both the display and the spectator: the human.

Humans are limited in their vision. This ranges from being unable to distinct colors at varying illumination scenarios to the limitation to focus only at one location at a time. Researchers have developed many applications to exploit these limitations to increase the quality of an applied compression. Some popular methods of vision-based compression are MPEG or JPEG. For example, a JPEG compression exploits the reduced sensitivity of humans regarding color and so encodes colors with a lower resolution. Also, other fields, such as auditive perception, allow the exploitation of human limitations. The MP3 compression, for example, reduces the quality of stored frequencies if other frequencies are masking it. For representation of perception various computer models exist. In our rendering scenario, a model is advantageous that cannot be influenced by a human spectator, such as the visual salience or saliency.

Saliency is a notion from psycho-physics that determines how an object "pops out" of its surrounding. These outstanding objects (or features) are important for the human vision and are directly evaluated by our Human Visual System (HVS). Saliency combines multiple parts of the HVS and allows an identification of regions where humans are likely to look at. In applications, saliency-based methods have been used to control recursive or progressive rendering methods. Especially expensive display methods, such as pathtracing or global illumination calculations, benefit from a perceptual

representation as recursions or calculations can be aborted if only small or unperceivable errors are expected to occur. Yet, saliency is commonly applied to 2d images, and an extension towards 3d objects has only partially been presented. Some issues need to be addressed to accomplish a complete transfer.

In this work, we present a smart rendering system that not only utilizes a 3d visual salience model but also applies the reduction in detail directly during rendering. As opposed to normal LOD-methods, this detail reduction is not limited to a predefined set of levels, but rather a dynamic and continuous LOD is created. Furthermore, to apply this reduction in a human-oriented way, a universal function to compute saliency of a 3d object is presented. The definition of this function allows to precalculate and store object-related visual salience information. This stored data is then applicable in any illumination scenario and allows to identify regions of interest on the surface of a 3d object. Unlike preprocessed methods, which generate a view-independent LOD, this identification includes information of the scene as well. Thus, we are able to define a perception-based, view-specific LOD. Performance measures of a prototypical implementation on computers with modern graphic cards achieved interactive frame rates, and several tests have proven the validity of the reduction.

The adaptation of an object is performed with a dynamic data structure, the *TreeCut*. It is designed to operate on hierarchical representations, which define a multi-resolution object. In such a hierarchy, the leaf nodes contain the highest detail while inner nodes are approximations of their respective subtree. As opposed to classical hierarchical rendering methods, a cut is stored and re-traversal of a tree during rendering is avoided. Due to the explicit cut representation, the *TreeCut* can be altered using only two core operations: `refine` and `coarse`. The `refine`-operation increases detail by replacing a node of the tree with its children while the `coarse`-operation removes the node along with its siblings and replaces them with their parent node. These operations do not rely on external information and can be performed in a local manner. These only require direct successor or predecessor information.

Different strategies to evolve the *TreeCut* are presented, which adapt the representation using only information given by the current cut. These evaluate the cut by assigning either a priority or a target-level (or bucket) to each cut-node. The former is modelled as an optimization problem that increases the average priority of a cut while being restricted in some way, e.g. in size. The latter evolves the cut to match a certain distribution. This is applied in cases where a prioritization of nodes is not applicable. Both evaluation strategies operate with linear time complexity with respect to the size of the current *TreeCut*.

The data layout is chosen to separate rendering data and hierarchy to enable multi-threaded evaluation and display. The object is adapted over multiple frames while the rendering is not interrupted by the used evaluation strategy. Therefore, we separate the representation of the hierarchy from the rendering data. Due to its design, this overhead imposed to the *TreeCut* data structure does not influence rendering performance, and a linear time complexity for rendering is retained.

The *TreeCut* is not only limited to alter geometrical detail of an object. The *TreeCut* has successfully been applied to create a non-photo-realistic stippling display, which draws the object with equal

sized points in varying density. In this case the bucket-based evaluation strategy is utilized, which determines the distribution of the cut based on local illumination information. As an alternative, an attention drawing mechanism is proposed, which applies the *TreeCut* evaluation strategies to define the display style of a notification icon. A combination of external priorities is used to derive the appropriate icon version. An application for this mechanism is a messaging system that accounts for the current user situation.

When optimizing an object or scene, perceptual methods allow to account for or exploit human limitations. Therefore, visual salience approaches derive a saliency map, which encodes regions of interest in a 2d map. Rendering algorithms extract importance from such a map and adapt the rendering accordingly, e.g. abort a recursion when the current location is unsalient. The visual salience depends on multiple factors including the view and the illumination of the scene. We extend the existing definition of the 2d saliency and propose a universal function for 3d visual salience: the Bidirectional Saliency Weight Distribution Function (BSWDF). Instead of extracting the saliency from 2d image and approximate 3d information, we directly compute this information using the 3d data. We derive a list of equivalent features for the 3d scenario and add them to the BSWDF. As the BSWDF is universal, also 2d images are covered with the BSWDF, and the calculation of the important regions within images is possible.

To extract the individual features that contribute to visual salience, capabilities of modern graphics card in combination with an accumulation method for rendering is utilized. Inspired from point-based rendering methods local features are summed up in a single surface element (surfel) and are compared with their surround to determine whether they “pop out”. These operations are performed with a shader-program that is executed on the Graphics Processing Unit (GPU) and has direct access to the 3d data. This increases processing speed because no transfer of the data is required. After computation, each of these object-specific features can be combined to derive a saliency map for this object.

Surface specific information, e.g. color or curvature, can be preprocessed and stored onto disk. We define a sampling scheme to determine the views that need to be evaluated for each object. With these schemes, the features can be interpolated for any view that occurs during rendering, and the according surface data is reconstructed. These sampling schemes compose a set of images in form of a lookup table. This is similar to existing rendering techniques, which extract illumination information from a lookup. The size of the lookup table increases only with the number of samples or the image size used for creation as the images are of equal size. Thus, the quality of the saliency data is independent of the object’s geometrical complexity.

The computation of a BSWDF can be performed either on a Central Processing Unit (CPU) or a GPU, and an implementation requires only a few instructions when using a shader program. If the surface features have been stored during a preprocess, a reprojection of the data is performed and combined with the current information of the object. Once the data is available, the computation of the saliency values is done using a specialized illumination model, and a priority for each primitive

is extracted. If the GPU is used, the calculated data has to be transferred from the graphics card. We therefore use the "transform feedback" capabilities, which allow high transfer rates and preserve the order of processed primitives. So, an identification of regions of interest based on the currently used primitives is achieved. The *TreeCut* evaluation strategies are then able to optimize the representation in an perception-based manner.

As the adaptation utilizes information of the current scene, each change to an object can result in new visual salience information. So, a self-optimizing system is defined: the *Feedback System*. The output generated by this system converges towards a perception-optimized solution. To proof the saliency information to be useful, user tests have been performed with the results generated by the proposed *Feedback System*. We compared a saliency-enhanced object compression to a pure geometrical approach, common for LOD-generation. One result of the tests is that saliency information allows to increase compression even further as possible with the pure geometrical methods. The participants were not able to distinguish between objects even if the saliency-based compression had only 60% of the size of the geometrical reduced object. If the size ratio is greater, saliency-based compression is rated, on average, with higher score and these results have a high significance using statistical tests.

The *Feedback System* extends an 3d object with the capability of self-optimization. Not only geometrical detail but also other properties can be limited and optimized using the *TreeCut* in combination with a BSWDF. We present a dynamic animation, which utilizes a Software Development Kit (SDK) for physical simulations. This was chosen, on the one hand, to show the universal applicability of the proposed system, and on the other hand, to focus on the connection between the *TreeCut* and the SDK. We adapt the existing framework, and include the SDK within our design. In this case, the *TreeCut*-operations not only alter geometrical but also simulation detail. This increases calculation performance because both the rendering and the SDK operate on less data after the reduction has been completed.

The selected simulation type is a soft-body simulation. Soft-bodies are deformable in a certain degree but retain their internal connection. An example is a piece of cloth that smoothly fits the underlying surface without tearing apart. Other types are rigid bodies, i.e. idealistic objects that cannot be deformed, and fluids or gaseous materials, which are well suited for point-based simulations. Any of these simulations scales with the number of simulation nodes used, and a reduction of detail increases performance significantly. We define a specialized BSWDF to evaluate simulation specific features, such as motion. The *Feedback System* then increases detail in highly salient regions, e.g. those with large motion, and saves computation time by reducing detail in static parts of the simulation. So, detail of the simulation is preserved while less nodes are simulated.

The incorporation of perception in real-time rendering is an important part of recent research. Today, the HVS is well understood, and valid computer models have been derived. These models are frequently used in commercial and free software, e.g. JPEG compression. Within this thesis, the *TreeCut* is presented to change the LOD of an object in a dynamic and continuous manner. No definition of the individual levels in advance is required, and the transitions are performed locally.

Furthermore, in combination with an identification of important regions by the BSWDF, a perceptual evaluation of a 3d object is achieved. As opposed to existing methods, which approximate data from 2d images, the perceptual information is directly acquired from 3d data. Some of this data can be preprocessed if necessary, to defer additional computations during rendering. The *Feedback System*, created by the *TreeCut* and the BSWDF, optimizes the representation and is not limited to visual data alone. We have shown with our prototype that interactive frame rates can be achieved with modern hardware, and we have proven the validity of the reductions by performing several user tests. However, the presented system only focuses on specific aspects, and more research is required to capture even more capabilities that a perception-based rendering system can provide.

Acknowledgements

This work covers the achievements and research I have performed during my time at the “Professur für Graphische Datenverarbeitung” at the Goethe University Frankfurt. However, this work would not be as it is now without helping hands.

First of all, I would like the give my gratitude to all my colleagues that always had a open door for conversation. The long discussions allowed me to develop the my theses and to solve issues during programming the prototypes. So thank you for all the proof reading, discussions, and brainstorming.

Furthermore, I would like to thank my supervisor and members of the committee who helped me to establish a scientific foundation. They allowed me to focus on my research and led the way for new topics and missing details: Detlef Krömker, Ralf Dörner, and Ulrich Schwanecke.

Finally, I would like to mention my parents, my brother, and my wife. The details and new views were refined and improved after hours of talking. The long days and nights of working would not have been possible without my wife who has supported me on this journey.

Contents

1. Introduction

1.1 Motivation

When watching movies, TV, or looking into magazines, everywhere we are confronted with photo-realistic images created by computers. What seemed to be unthinkable half a century ago is, due to the quickly emerging capabilities of computer technology, now a simple task. But, still large computational power is needed for real-time rendering of photo-realistic images. One of major challenges in 3d rendering is to reduce the flood of information without loss of detail detectable by a human observer.

The decision, which features of an image can be omitted, is usually made a priori, and the rendering system is adapted by selection one of the predefined levels. However, the great heterogeneity between current systems, e.g. a high performance computer with a dedicated graphics card as opposed to a smart-phone, does not allow to make this decision in advance. With the computational power of modern computers as well as the increasing capabilities of graphics cards, an implementation of real-time decision making is at hand. Therefore, a smart rendering system is needed that utilizes the available resources most efficiently. Based on the before mentioned systems, the high performance computer would generate photo-realistic images while the smart-phone displays the complete 3d object with the highest detail possible and preserves interactivity at the same time.

To create high detailed images, an approach should take into account for whom those images are created. This would allow a rendering system to avoid unnecessary calculations as the Human Visual System (HVS) is limited in its ability to perceive details. These limitations arise from various factors, like the laws of physical optics, the visual pathway or the fact that a human observer can only focus at a single location at a time. By exploiting these limitations, important regions of an object could be handled with more care while unimportant ones could be represented as an approximation. This reduces rendering or processing time and preserves visual quality. Looking at the HVS, either early vision or higher visual processes serve as a starting point for such an approach.

The early vision offers several aspects that may be exploited, including color appearance, foveal focusing, as well as motion perception. The latter, for example, creates the impression of moving pictures in computer graphics and movies. This is shown in a flip book, which contains a set of images. Its images differ in a small amount regarding to their predecessor and successor. Changing these with more than 10 pictures (or frames) per second creates the impression of a smoothly moving object in the flip book[1]. Other phenomena include the placement of cones and rods in the retina of the human eye, which restricts the range of colors or the change of illumination perceived. This is usually leveraged by image or movie compression methods, such as JEPG or MPEG. In the case of image

[1] If sketched carefully.

compression with JPEG, one part is to exploit the reduced sensitivity of humans regarding color, and store color information with less quality.

Higher visual processes can be influenced by tasks performed by the observer because humans can force themselves to look at a distinct direction or concentrate on a specific detail. This can lead to a phenomenon called "inattentional blindness" where spectators may not see objects right in their foveal center because they concentrate on fulfilling a task. This allows to remove more details of an object without being noticeable or perceivable. The direction of gaze also provides useful information and allows to perform a selection during rendering: Objects that are out of sight have their detail reduced as long as the spectator is not gazing towards the object's location. This way large compression of the data that will be displayed can be achieved.

Many researchers evaluate visual perception and incorporate these into the rendering process. Yet, their approaches lack the possibility to acquire perception information in a real-time application or to apply this information to a single object within a scene. Most of these methods are just not suited for real-time applications as they require a comparison image for evaluating perceivableness. When accounting for higher visual processes, the selection is dependent on human input and acquisition of relevant data is costly. For example, to extract the gaze direction, an eye tracker with high sampling rate is required, and this additional hardware is usually not available. Furthermore, an imposition of a task to a spectator is hard to achieve and can never be assured.

Therefore, we focus on alternative methods that allow perceptual evaluation during run-time using an early vision model. Image or movie compression methods often include perceptual processing, but are not directly applicable to rendering. However, they have been validated often and are widely accepted and used. As opposed to the higher visual processes, they also do not require task prediction or additional hardware. Therefore, these are well suited for any computational system, and we assume that an application of a perception-based compression on the source data, i.e. the objects, is reasonable. On this behalf, some researches have been done, and the results are presented within this thesis.

1.2 Definitions

In this work, some notions and definitions are used without any further explanation or are shortened for ease of writing.

When talking of perception in this context, the visual perception of humans is meant. Perception in its basic meaning also includes auditive, sensitive, smell-, and taste-related perception. As this is not relevant in our vision-based scenario, these other senses are not taken into account.

The creation of digital images, arising from a 3d scene description, can be performed with different rendering techniques. There exist various methods to obtain the final images, and usually they are based on either the so called rendering pipeline or raytracing. Within this work, both terms will be used frequently, but their principle will not be introduced except for crucial parts. A basic knowledge of these algorithms common within computer graphics is expected. For further and more detailed

explanations, the reader is referred to an introductory book, such as [SMA10].

1.3 Thesis

Perception-based rendering can ameliorate the overall processing time without introducing visible artifacts. Artifacts or defects may be generated by reducing the detail of the rendered object. Yet, the HVS may not be able to detect them in both real-time applications and images if the reduction exploits its limitations.

Only a few special applications in computer graphics can benefit from the existing approaches. As opposed to other perception-based methods, our focus is on real-time applicability. This is often discarded as direct processing of perceptual information requires large amount of computation.

This lack will be addressed, and solutions for this problem will be presented and discussed throughout this thesis. In chapter ‘Theses’ on page 63 a more refined definition of the theses will be given.

2. Background

In this section, a short overview over common hardware and software for rendering and perception-based approaches will be presented. This information is relevant to understand how the framework of our thesis works and how perception-based methods profit from modern hardware.

With the term rendering, we mean the processing of three dimensional (3d) data to gain a two dimensional (2d) image. This image is usually presented within a graphical user interface (GUI) on a monitor.

For animations or moving pictures, the flip book principle creates the impression of smoothly moving objects. This continuous impression of motion is connected to the critical flicker fusion frequency, and it is individual for each human. Furthermore, it is also influenced by the light scenario [Lue03]. Usually, a frequency is chosen to be above the average fusion frequency, which is about 10 - 16 Frames Per Second (FPS), and thus no flicker is perceived. For example, motion pictures offer 24 FPS while computer games try to achieve at least 30 (or even 60) FPS. A frame represents a final image that is displayed by an output device.

2.1 3d Rendering

Two main approaches to generate an image of a 3d scenery exist. Namely the rendering pipeline, e.g. the projective methods, and raytracing – or raycasting – techniques.

The first is directly supported by modern graphics cards where all so called primitives, e.g. a triangle or a quad consisting of vertices, of a 3d object are processed with the same transformation information. As these primitives are sequentially processed and the same operations throughout the complete rendering process are applied, the term pipeline is used. This pipelining allows great optimization and directly scales with the number of primitives provided. Therefore, the time complexity is connected to the number of primitives provided because each primitive will be processed by the pipeline. By projecting these primitives onto a 2d plane, a final image is composed. The interpolation of the primitives with a scan conversion algorithm then reconstructs a filled object. To assure only the top most primitives are drawn, a depth check needs to be performed, for example, by utilizing the z-buffer algorithm.

Acceleration structures allow to reduce the number of primitives rendered without affecting the visible primitives in a given scene. Culling methods, such as backface-culling, offer a great performance boost. By omitting all backfacing primitives, the number of primitives to be drawn is reduced (on average) by half. The main drawback of this approach is that an overdraw still occurs and cannot be avoided because during processing no information is available whether a primitive will be visible in the composed image. The primitive will be processed, but may finally be rejected by the z-buffer.

Therefore, it does not contribute to the final image.

Another way to generate an image using 3d data is to apply raycasting or raytracing. Instead of projecting primitives onto a plane, which then is displayed, the plane is directly sampled, which generates a ray, and a search for intersecting primitives is performed. Rather than depending on the number of primitives, raytracing scales with the number of samples used and the number of intersection calculations. With acceleration structures, intersection resolving can be greatly improved. This results in a logarithmic scaling in relation to the number of objects present in the scene. In the long term, raytracing will become more performant in comparison to the renderpipeline approach because of the increasing number of primitives. Despite this fact, current Graphics Processing Units (GPUs) offer great benefits for processing geometry or image data. For this reason, an introduction to the basic operations is given.

2.1.1 Graphics Processing Unit

In recent years, the processing capabilities of the graphics cards have evolved dramatically. Starting with a fixed pipeline, modern graphics cards possess freely programmable processing units, the so called GPUs. These allow not only high performance calculations in graphical means, but also are usable for general purpose calculations. In this context General Purpose Graphics Processing Unit (GPGPU) is a common term.

These GPUs differs from current Central Processing Units (CPUs) because the graphic cards are optimized for pipeline operations and utilize the rendering pipeline approach. Each 3d primitive is processed equally to generate the visual output for the display. These primitives usually have a single four valued vector for position, which represent its location in a homogenized coordinate system.

To process the provided data at once, the graphics card hardware builds upon a streaming Single Instruction Multiple Data (SIMD) architecture. Combining the streaming SIMD architecture with non graphic related operations offers a high performance calculation capabilities as long as the data is encoded in a suitable format, e.g. a correct data alignment. An example GPGPU application is a physical simulation because vector and matrix operations are already implemented. Also, other applications, such as an AES encryption or image manipulations, are possible [Fer04; PF05; Ngu08].

Along with the programmable rendering pipeline, different stages were defined. These depend on the underlying shader model and are strongly connected to the DirectX™[1] framework. As this framework is limited to Windows™platforms, openGL is mainly used when software is to be distributed to various operating systems. Usually, both offer the same features, but for the sake of simplicity only the Direct3D™equivalents are presented.

Shader model 1 introduced basic vertex modification and picture element (pixel)[2] processing. When it was proposed, these could only be manipulated using assembler instructions. Shader model 2 and 3 extended the GPU capabilities by increasing the instruction count and adding various functions,

[1] DirectX™ and with Direct3D™ provide all necessary parts for game development within a single framework

[2] In openGL the individual pixels are called fragments. This distinction arises from the fact that fragments hold more information than a simple color as pixels do.

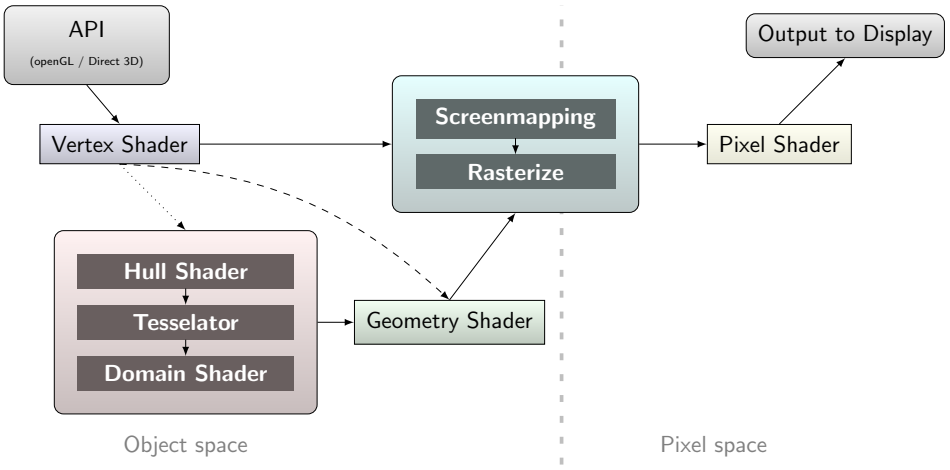


Figure 2.1: The basic pipeline when using graphic cards for rendering. The geometry shader has been added in shader model 4 and the tessellation shader was introduced in version 5. Both are optional within the pipeline and are activated only if needed.

which were either directly implemented on the hardware or were efficiently mapped to built-in functions. When programming shaders, these functions should be preferred over hand written methods whenever possible as the hardware supports these directly.

Shader model 4 added a new, separate stage to the rendering pipeline. As it is a designated shader to create geometry, it is called geometry shader. Using either procedural or texture related information, new geometry can generated during processing of a vertex shader. This allows to dynamically refine any given object. Also the Unified Shader Model was introduced, which removed various limitations of the vertex and pixel shader. The Unified Shader Model provides equal processing capabilities in all shaders stages and allows the graphics card to distribute its individual processors more efficiently.

With shader model 5, the compute shader and tessellation shader were introduced and are directly available in the graphics card hardware. The tessellation stage allows continuous Level Of Detail (LOD) calculations directly on the graphics card without the need to define them in advance. This is a triangle-based refinement process and so differs from the geometry shader, which generates new and independent vertices. The compute shader is specialized for general purpose computation, and it does not depend on the rendering pipeline, like vertex, geometry, tessellation and pixel shaders. It also adds random memory access on the graphics card and so allows more advanced GPGPU implementations. The individual stages and the shaders introduced at those stages are shown in figure 2.1.

To write and design shader programs, various methods are available. One can either generate assembly code for the graphics card or utilize a high-level language. OpenGL offers openGL Shading Language (GLSL) while Direct3D^{TM}offers High Level Shading Language (HLSL). The former is strongly related to the openGL framework while the latter was defined in cooperation with nVidia^{TM}. In this joint venture with nVidia^{TM}both C for Graphics (CG) and HLSL were defined. While HLSL is bound to Direct3D^{TM}, CG can also be compiled for openGL. Both offer the possibility to create sets of shader programs. These so called techniques assure that the correct shaders are executed

together [FKF06].

2.1.2 Mesh-based Rendering

The common approach to represent objects or surfaces for rendering are meshes of vertices, which are composed to primitives. These primitives are processed and rasterized, so that a filled surface is generated. Usually, they are decomposed to triangles because rasterization and interpolation of triangles is efficiently performed with barycentric coordinates.

The barycentric coordinate system describes the complete surface of a triangle with 2d coordinates. Providing the vertices of a triangle, each surface point can be determined using formula (2.1) (taken from Shirley et al. [SMA10]).

$$P = \alpha \cdot \vec{a_1} + \beta \cdot \vec{a_2} + \gamma \cdot \vec{a_3} \tag{2.1}$$

The parameters of the vectors are subjected to $\alpha + \beta + \gamma = 1$. Then, the point P is inside the triangle iff $\alpha \geq 0, \beta \geq 0, \gamma > 0$. The barycentric coordinates allow to interpolate each sample and allows to calculate positions, color, and surface normal information.

The primitives are provided to the rendering pipeline (refer to figure 2.1) where the vertices are transformed into clip space and rasterized using barycentric coordinates. Finally, shading is applied using an illumination algorithm. Usually, a simple Phong or Blinn illumination is used, but more sophisticated methods exist to create even more realistic images. These shading algorithms require interpolated surface information, such as normal vectors or texture coordinates. The latter is needed to allow a texture to be applied to the triangle as a texture yields more visual accurate results than plain illumination alone. Normally, both methods are combined to achieve the final pixel color value, which is written in the framebuffer and displayed by the operating system.

An advantage of using a mesh of triangles is that it can be subdivided to create a more detailed representation of the surface. As the triangles shrink in size, they will fit the surface more accurately. In combination with a Bézier-patch, a smooth, highly detailed representation of the surface is derived. With today's hardware (refer to section 'Graphics Processing Unit'), a hardware tessellation is possible that allows an on-the-fly refinement of any given mesh. This leads to a dynamic Level Of Detail (LOD)-representation limited only by the graphics cards processing capabilities instead of a designer-based identification or preprocessing approach.

A mesh representation is not limited to a single base type, but all have in common that the represented surface is convex and does not contain any cavities. When these requirements are met, each base type is decomposable into multiple triangles. Triangles have several advantages over universal quads or n-sided polygons. All of the three vertices share a common plane, and the surface is correctly interpolated with barycentric coordinates. Additionally, when using only one basic primitive type, optimizations inside the rendering pipeline can be performed because each algorithm can be specialized to process this particular type.

As with any other surface representations, mesh-based representations do not cover the inner

structure of an object. Therefore, modeling of cracks or the simulation of volumes comes with a high penalty as the underlying data has to be calculated on-the-fly. Another drawback of mesh-based representations is that with higher detail more and more triangles will degenerate. When vertices will get close to each other, they will be projected onto the same pixel in the rendering pipeline, which causes overhead during rasterization and vertex processing. Both the overhead and the missing volume information are reasons why other representation methods are still in use.

2.1.3 Volume-based Rendering

Volume elements (voxels) have become an important representation type because of the increasing need to visualize the large data available from modern CT scans and MRTs. Instead of the surface of an object, the complete volume is modeled, and thus it allows to create correct volumetric effects, like volumetric scattering. Furthermore, transparent or translucent objects can be created because the density of a single voxel is stored and accumulated during run-time.

Basically, one has to differentiate between data acquisition and rendering. While the former is done using the before mentioned methods (CT, MRT, etc.), for the latter multiple approaches exist.

In case of the rendering pipeline, a polygonal surface is reconstructed by representing voxels as vertices or triangles. With the combination of marching cubes [LC87], a correct layout of a triangle within the voxel can be determined. With this method it is also possible to extract an iso-surface by precalculating a set of polygonal configurations. As the original algorithm was patented, marching tetrahedrons were developed to circumvent this issue [NY06]. Both algorithms generate a set of polygons, which are drawn to create a visual output. Another visualization approach is splatting, which is commonly used for point-based representations. By assigning an extend that is based on the volume's size to a voxel, it is drawn using either circles, ellipses, or any other primitive type, such as cubes or other 3d objects.

On the one hand, a high sampling rate is required to create smooth surfaces without cavities in the representation. On the other hand, this has an great impact on the final data size because a voxel-grid is usually uniformly sampled. Therefore volume representations gain great benefit from acceleration structures, such as octrees. These search structures increase selection performance when generating the display output, and consequently optimize the overall performance of the rendering process. Current implementations use the GPU to calculate the required data solely on the graphics card, avoiding processing on and transfer from the CPU [Cra+09].

One of the major drawback of volume representations is that all information, whether it is usable or not, is sampled and stored individually. This may result in large data waste as many voxels do not contain valuable information. The selection of the data layout has a large impact on the final space consumption. If the data layout allows compression, unused or empty voxels are combined or discarded to reduce the data overhead [GP07]. This of course is an approximation of the base data and may introduce artifacts.

2.1.4 Point-based Rendering

Point-based rendering is a very naïve approach to generate images because data is directly represented without any topological information. By providing a point-size, it is possible to transform each sample and display it using either raytracing or the rendering pipeline. As with mesh-based rendering, the representation is limited to the surface, but here no adjacency information is available.

This is the main advantage, but also the major drawback of point-based rendering methods. Without the adjacency information, no interpolation of a surface is possible. So, the surface must be sampled with high density to assure that no cavities are introduced during resampling. This requires large amount of data, and points may overlap or will not be visible in the final frame after all. But, with today's polygon count, many triangles degenerate, and point-based rendering becomes a more valuable representation. By displaying a single point instead of a triangle the polygon count is reduced by the factor 3.

Another reason for the application of point-based rendering is that modern scanning techniques provide plain 3d positions without any connection information. The reconstruction of these is very time consuming, and its correctness cannot be assured in every situation. In such cases, point-based rendering provides direct rendering and avoids the need of precalculation.

Rendering point-based data sets can be performed with both the rendering pipeline and raytracing. But, with raytracing or lower sampled data sets, problems arise. Holes may appear where no points are found. This is avoided with the application of reconstruction methods, such as *Point-Set-Surfaces* presented by Alexa et al. [Ale+03], which create a mathematical representation of the sampled surface. Raytracing then operates on this implicit representation of the surface [AA03]. If sampled data is used instead, an extend is given to each point, and a closed surface is reconstructed by increasing the point-size when gaps appear [RL00; BSK04; Zwi+02]. This rendering method is referred to as splatting because the points are figuratively splatted onto the frame. The increase of the extend, however, is an approximation of the surface, and its quality can be enhanced with circles or ellipses, which reduce alias effects during resampling [Zwi+02].

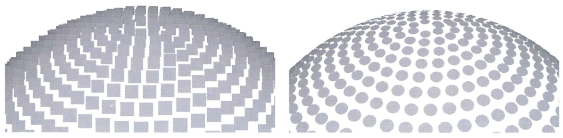


Figure 2.2: The perspective correct splatting method. In the left image the uncorrected version with splatted quads is shown. The more accurate solution in the right fixes the splat orientation and applies a elliptical splatting.

Rendering quality is enhanced when providing additional surface information, such as normals or texture coordinates. This allows texturing and more sophisticated illuminations. For distinction, these points, inspired by pixel or voxel, are then called surface elements (surfels). These surfels are evaluated by a pixel shader to create the image. By projecting the surfels onto the surface with the help of their normals, a perspectively correct surface can be reconstructed. This results in images of higher quality and a more accurate surface representation (refer to figure 2.2).

The interpolation capabilities of the point-based representations, e.g. their topological independence, allow interactive refinement during rendering and dynamic transfer of the data [PKG06].

Rendering of point data or point clouds greatly benefits from current GPU capabilities by direct computation on the graphics card. When streaming a sequential hierarchy, the graphics card controls the visible output instead of relying on a CPU-based computation. This results in higher frame rates because the bottleneck of data transmission between CPU and GPU can be omitted [DVS03].

2.2 Visual Perception and the Human Visual System

Vision in general depends on light, and at least one light source is required for the HVS to work. This light, which is emitted by a so called illuminant, e.g. in natural scenarios this illuminant is usually the sun, travels through the air and is reflected by one or more objects until it finally reaches the eye. Every time an object is hit, the light ray is scattered, reflected, or absorbed. Additionally, the light ray may be altered, e.g. regarding its color or intensity (energy). With the reflection or scatter, indirect illumination is created. The travel is continued until its energy has been completely absorbed, e.g. when it hits the eye.

Visual perception of humans is influenced by various factors, and different approaches have been made to formalize the HVS. It is common to separate the visual system into two processing paths, low- and high-level vision. Low-level or early vision includes the visual pathway from the eye to the visual cortex V1. High-level vision includes tasks, e.g. to count one type of an object in the scene, and spectator related behavior. In other words, tasks that are influenced by spectator and so determine, which portions of a scene receive special attention. We will refer to high-level as higher cognitive processing. Due to the focus on certain aspects of a scene, others will be neglected, a phenomenon known as “inattentional blindness”. Exploiting this and other phenomena, allows us to create a rendering method where individual objects of a scene are fitted to the spectators level of attention. However, the required conditions have to be imposed correctly, and tasks that result in “inattentional blindness” must be determined a priori to assure correctness of the approach [CCW03].

Before presenting approaches for modeling the visual perception, we need to define the HVS to establish the necessary knowledge required to understand these models. This includes early vision as well as higher cognitive processes of the visual pathway. Once these processes have been identified, we derive a computer model that simulates the HVS based on the input provided by an image. This additional perceptual information will be used in a rendering context to control the appearance of either individual objects or the overall scene.

For a more detailed introduction into the HVS and the processes performed, the reader may be referred to excellent surveys, such as presented by Ng et al. [NBZ07] or Lennie and Movshon [LM05], or books specialized on vision or visualization, such as “Information visualization” from Ware [War00], Frisby’s “Seeing” [FS10] or “Visual Perception” from Cornsweet [Cor74].

2.2.1 The Human Eye

The eye can be understood as a camera. Like a camera it has a lens, an aperture, and a film, i.e. the retina. The lens allows us to follow or see different objects in the scene by moving it with the attached muscles or focus nearer or farther objects by deformation. Due to the optic laws, the lens creates an inverted picture of the world from the eye's position onto the retina. The iris adjusts the aperture by enlarging or narrowing itself, and thus limits the amount of light passing to the retina. So, we adapt to different light situations, for example, to a dark night when only a small amount of light reaches the retina as well as to a bright day where the aperture is reduced to avoid damage to the retina. The transition between these two extremes requires some time. An adaptation to bright light occurs faster than the opposite way.

The seen visual pattern may be described based on the extends defined by both lens and eye. The notion *visual angle* is defined by the angle that is "[...] subtended by an object at the eye of an observer" [War00]. The visual angle depends on the objects size and the distance to it. As a rule of thumb, a thumbnail seen at arm's length has a visual angle of approximately 1 arc degree.

Photorecptors of the Human Eye

The light rays, which are electromagnetic waves, are absorbed by the photoreceptors, and the light is converted into an pattern of energy, which is processed by all stages of the HVS. Physically speaking, each wave has a specific frequency or wavelength that specifies the optic behavior and energy. The spectrum that the human eye is capable of perceiving is limited only to a small range of wavelengths, starting at approximately 400nm and ending at approximately 700nm [War00]. An example where the complete visible spectrum is seen, is a rainbow. The violet colors correspond to the shorter wavelengths, i.e. 400nm, and the red color at the other side of the rainbow maps to higher wavelengths, i.e. 700nm. Shorter wavelengths i.e. ultraviolet light and longer wavelengths i.e. infrared light [3] cannot be perceived by the human eye.

Once the light passed the lens and the iris, it travels through the *corpus vitreum* or vitreous humor and finally hits the retina. The retina consists of an array of photoreceptors. These react on the incoming light rays in two different ways depending on their cell type. The first cell type are the rods, which are highly reactive in low light scenarios, but only provide luminance information. In a daylight scenario, they are overloaded and produce no viable visual information. For example, photoreceptors of nocturnal animals and most mammals are mostly comprised of rods [FS10], but have different sensitivities to preserve the ability to see at daylight and dark night. The other type are cones, which provide color information, but are less sensitive. In average, a human eye has 100 million rods and 6 million cones. In figure 2.3 the distribution of rods and cones is shown.

When it comes to model the photoreceptors of our eyes, both cones and rods should be taken into account. In a daylight scenario, as stated before, the rods are overloaded and produce no viable visual

[3] This naming convention, ultra for above and infra for below, arises from the frequency-based representation of the waves

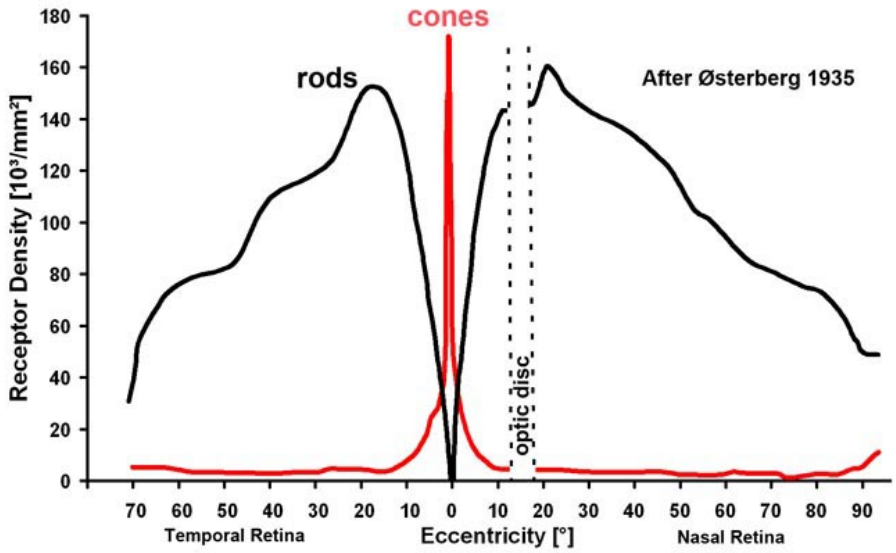


Figure 2.3: The distribution of rods and cones over the retina. Most of the cones are located in the center region, called the fovea. The region without any cells is called the blind spot as here the arteries and nerves enter the eye. Adapted from `http://webvision.med.utah.edu/book/part-viii-gabac-receptors/light-and-dark-adaptation/`. Website visited 22.08.2011

information. For this reason they are often omitted by most researches [War00].

The retina consists of three cones types. Each of them generates a different response depending on the incoming light's characteristics, i.e. the wavelength. These three receptor types, the s(hort) reacting on short wavelengths, i.e. blue colors, m(edium) and l(ong) responding to long wavelengths, i.e. red colors, are not distributed evenly and occur at varying frequencies.

By performing tests to acquire individual responses of the photoreceptors, their sensitivities are extrapolated to result in a mathematical description unique to each type. In figure 2.4, a plot of the acquired data is shown. The results are taken from a study performed by Stockman and Sharpe [SS00] that measured the short, middle, and long cones responses.

Ganglion-Cells

The cones are connected through ganglion-cells, and the information exchange between them is governed by the opponent process as described by Ewald Hering's model [War00]. The process is based on black-white (or luminance), red-green, and blue-yellow color differences derived from the original signals provided by the cones. It also explains why there is no color like blueish yellow or greenish red. The process is depicted in figure 2.5.

The ganglion-cells receive input from the three cone types and either add or subtract these from each other. The luminance or black-white axis is derived by adding the signals present from the L and M cones. This is an adaptation of the original definition where L+M+S is calculated.

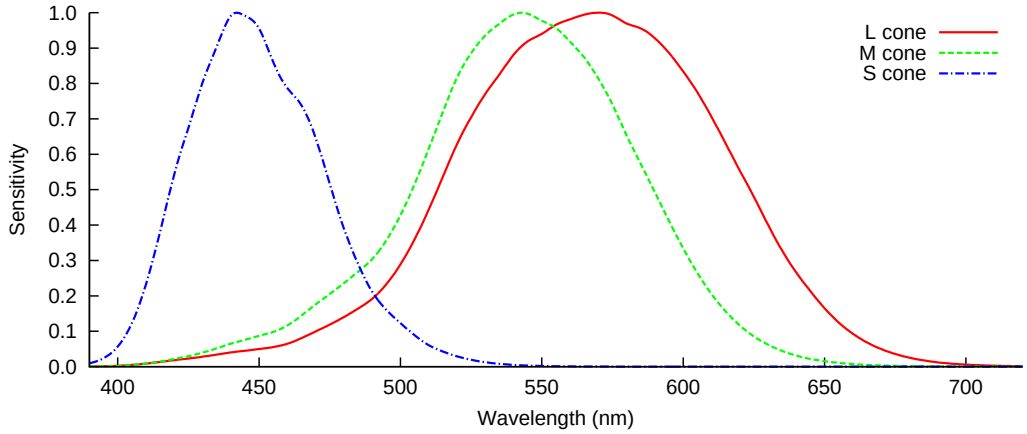


Figure 2.4: The response curves of the three cone types (S,M,L). The short have a high peak in the blueish colors and are the most sensitive ones. The medium and the long cones almost overlap and account green to red colors. The cone response values are normalized and are taken from Stockman and Sharpe [SS00].

The red-green difference is given by subtracting the M-cones signal from the L-cone input. The last difference requires an intermediate result, namely the color yellow. This is derived by first adding the L-cone and M-cone signals resulting in the yellow color signal. Then, the S-cone input is subtracted from this signal, which defines the third axis.

Two types of ganglion-cells are involved in color discrimination:

1. Midget ganglion-cells have a high spatial resolution while responding slowly to changes in intensity. This means a large area is processed accurately, but the time resolution is low.

2. Parasol ganglion-cells have a high temporal resolution and require less cone signals [NBZ07]. This type is mainly found in the periphery and make up 10% of all ganglion-cells present.

For the sake of completeness, the bistratified retinal ganglion-cells need to be mentioned, but their function is not well understood [LM05] because they have been discovered just recently.

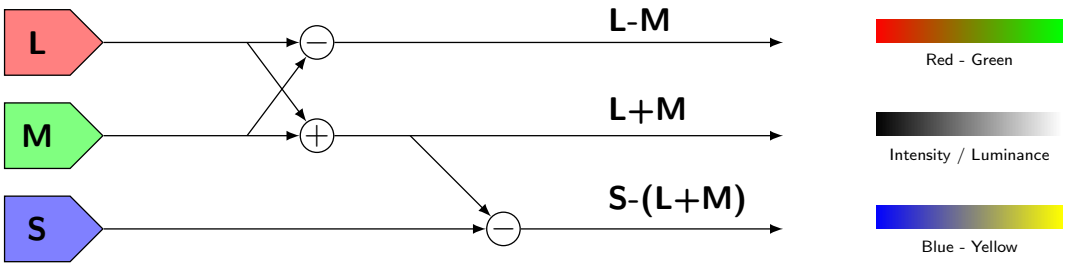
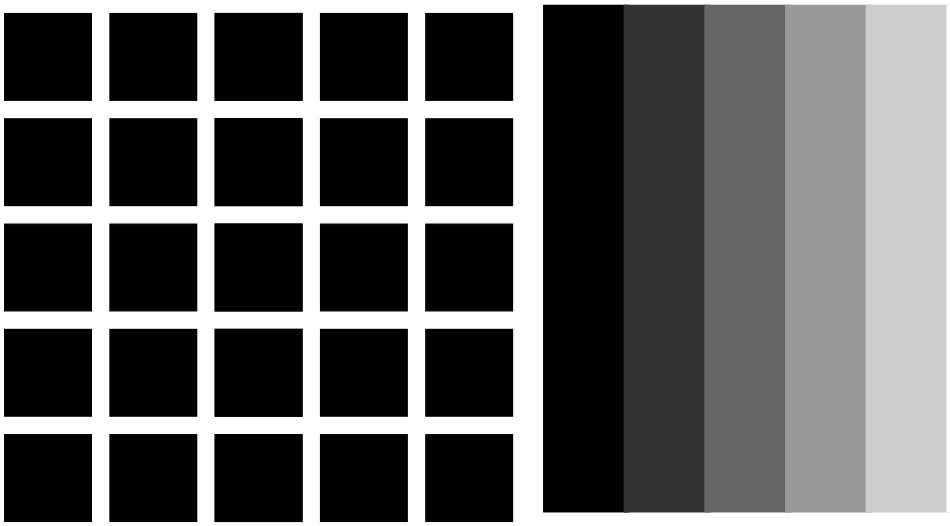


Figure 2.5: The opponent process model as defined by Hering. The input signals created from the cones (S,M,L) are processed by the ganglion-cells. Depending on the applied operation, a three axis color system is created. For the Y-B path, an intermediate result, the yellow color, is required before the subtraction can be applied. Figure adapted from [War00, p. 119].



(a) A Hermann grid. The black dots appearance results from less inhibition at that position.

(b) Mach bands created by illumination discontinuities. The left of each bar appears lighter than the right side.

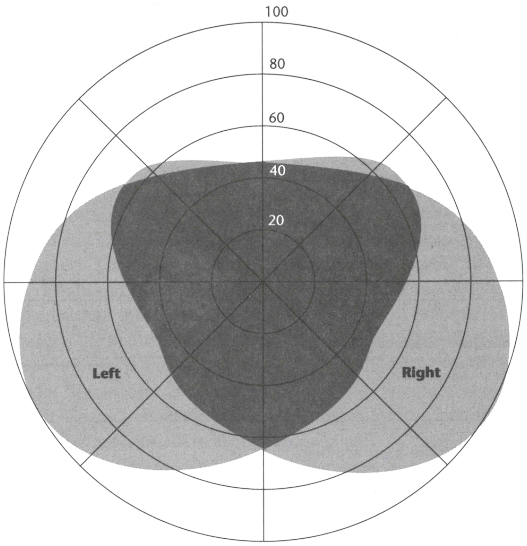
Figure 2.6: Different illusions, which are explained by the Difference Of Gaussian (DOG) model. Images taken from [War00].

Receptive Fields

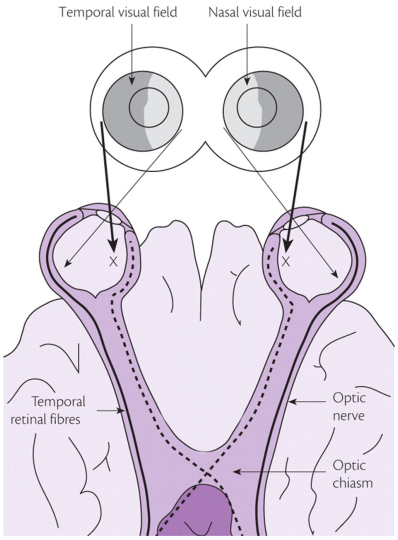
Based on Ware's [War00] definition, the ganglion-cells in the retina are organized with circular receptive fields. These build the visual area over which a cell emits a signal if light falls on it. There are two possible configurations for the ganglion-cells: *on-center* and *off-center*. Depending on the stimuli, the cell will emit a different signal rate. When the center is stimulated, the signal rate increases, it is excited. It decreases if the *off-center* is hit, and the signal is inhibited.

A mathematical model describing the excitation and inhibition is given with the Difference Of Gaussian (DOG) model. In the model, one cell represents the center while the other represents its surrounding. This allows to simulate excitation or inhibition based on the receptive field that has been stimulated.

A large number of illusions correlated with inhibition are explained with the help of the DOG model, such as the Hermann grid or Mach bands (see figures 2.6a and 2.6b). The black dots in the Hermann grid appear at the intersections because there is less inhibition from the surrounding boxes (see figure 2.6a). The Mach bands occur due to a discontinuity of the brightness. These discontinuities are accentuated by the HVS, and an application of the DOG model inherently reproduces this overestimation of the signal. The Gaussian does not have any discontinuities and only changes gradually. In the example figure 2.6b, the left side of each bar appears lighter than the right, but the intensity does not change inside each bar.



(a) Human visual field. The light gray areas provide only monocular vision while in the dark gray areas binocular or stereo vision information is available. Taken from [War00].



(b) In the optic chiasma the acquired signal are swapped, so that left and right parts of the scene are assigned correct. Adapted from [WCF10].

Figure 2.7: The visual field of a human and the optic chiasma, which generates it as the signals of both eyes are exchanged.

Visual Field

In the area called fovea, the cones are densely packed. Here, vision is sharpest, and the most amount of detail is perceived. The visual angle of the fovea is about 1.5 to 2 arc degrees, and the acuity rapidly decreases with increasing visual angle. The visual field that a human spectator sees when gazing straight ahead is depicted in figure 2.7a. The regions where both eyes overlap is shown in dark gray, and there binocular information is available. The other regions shown in gray only provide monocular information.

2.2.2 The Optic Chiasma

The opponent signals of each eye are sent through the optic nerve crossing at the optic chiasma. At this point, the nasal parts of the visual field are swapped to the opposite sides of the brain while the temporal parts remain on the same side (see figure 2.7b). So, a complete image for each visual field is recreated before processing of the signals is continued in the Lateral Geniculate Nucleus (LGN).

2.2.3 Lateral Geniculate Nucleus

The visual signal is relayed by the LGN to the primary visual cortex. The neurons of the involved cells in both LGN and V1 are anatomically well separated regarding the type and functionality [NBZ07].

In the LGN, two major parts are distinguished. One for the chromatically opponent neurons, and the other for the broadband neurons. A broadband neuron is defined as a cell that is excited in its center

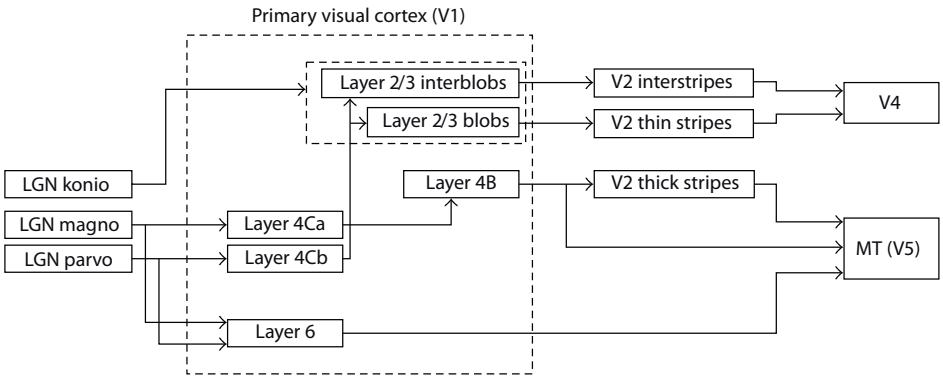


Figure 2.8: The connections of the individual LGN processing paths to the V1. MT denotes the middle temporal area of the visual cortex. Figure taken from [NBZ07].

by any wavelength while being inhibited by same wavelength in its surround. The LGN consists of 6 layers with three different cell types.

The parvocellular cells (P) are driven by the midget ganglion-cells in the retina and represent the upper four layers of the LGN. The P-cells process color signals and extract both color and form information [NBZ07].

The magnocellular cells (M) are not sensitive to single colors, but are tuned to an overall luminance value. They process the input of the parasol ganglion-cells and accordingly respond on luminance changes [Gos96].

The last type of cells in the LGN are the koniocellular cells (K). These have recently been discovered, together with the bistratified ganglion cells. They are located between the independent M and P layers in the LGN [LM05]. The K-cells process the input of the S-cones, but their properties have not been well characterized, yet [LM05].

2.2.4 Visual Cortex

The processed signals are forwarded to the visual cortex where higher cognitive processes take place that steer our attention to detail. Following the Brodmann notation, it is located in area 17. As it is visually identifiable by stripes – caused by the myelineated neurons –, it is also called the striate cortex [NBZ07].

The individual processing paths are depicted in figure 2.8. It shows the connections of the individual processed signals from LGN to V1. The interconnections in V1 and the projections towards the upper areas of the visual cortex are depicted as well.

The V1 is separated into 6 layers by splitting it based on the relative density of neurons. The cells in both V1 and V2 are tuned each for different properties. This includes orientation, size, color – separated in two signals –, stereoscopic depth and motion [War00]. With the orientation, our visual system performs a segregation of objects based on their border, which supports the identification of single objects in the complete image. The orientation is evaluated from different sets of cells, and each of

them is aligned at different angles that fire when the input matches their individual direction [War00].

There is also a feedback path between V1 and LGN, allowing the LGN neurons to become orientation selective despite the fact that the neurons themselves are not capable of doing so [NBZ07]. Ng et al. also state that "the function of the LGN is poorly understood beyond being viewed as a relay station" [NBZ07].

2.2.5 Depth Cues

Human visual perception is specialized on operating in 3d environments. In order to operate efficiently in such an environment, we need to be able to estimate the depth of a scene. Our ability to judge the distance to an object, however, does not rely on stereoscopic features alone. Other features of the scene provide depth cues as well, and most of these cues are extracted from a single image. People may be unable to perceive stereoscopic depth without noticing it at all.

Ware [War00] declare these cues according to their production method: Monocular and binocular. He gives a list of *cues*, which is presented below. Other researchers may cover a different or more complete list, but we will stick with the given definition. Ware states that monocular cues are those that are available from an image:

- Linear perspective
- Texture gradient
- Size gradient
- Occlusion
- Depth of focus
- Cast shadows
- Shape-from-Shading
- Structure-from-motion (requires moving pictures)

The binocular cues require two eyes to be correctly determined and are defined as follows:

- Eye convergence
- Stereoscopic depth

The effective ranges of these cues have been empirically studied by Cutting and Vishton [CV95]. One of their results is a distinction of three classes around the observer: personal, action and vista. Each class weights the *cues* differently depending on the distance to the viewer. This includes that some of the *cues* may not be visible to the observer at a certain distance and are not accounted for. Cutting and Vishton introduce a ranking of these cues. For example, in the personal space, which is around 2 meters of one self, relative size is not as important as binocular information [CV95]. The most important cue is occlusion, which provides useful information at all tested distances.

2.3 Model of the Human Visual System

A solution for perception-based rendering either has to influence higher cognitive processes, e.g. by inducing a task to the spectator forcing "inattentional blindness", or account for early vision. It would also be possible to simulate the complete processing performed in both. We intend to apply a model for early vision because we do not want to imply restrictions to the scenario by enforcing tasks or the need to predict spectator behavior.

When a model does not simulate higher cognitive processes, it is referred to as a bottom-up model. Such a bottom-up model processes the input, e.g. an image or a scene, and provides the results usable for higher cognitive processing. When only higher cognitive processes are involved instead, i.e. by asking the observer to look at a special location, the model is referred to as top-down or task-driven.

2.3.1 Contrast Sensitivity Function

When examining the visual process along its path from the retina to the visual cortex, the main operation is based on a difference calculation of the provided signals. As the system operates with analog signals, a threshold value is incorporated within the HVS to assure that only large differences will be accounted for. These thresholds or sensitivities are imposed by biological and chemical properties that limit the capabilities of human visual perception, and a universal representation is derived from these. This representation, however, assumes that no defect in the visual system is present.

Sensitivities of the Human Visual System

The HVS provides three layers of sensitivity: Light level, signal content and spatial frequency [Dal93]. The last is expressed with a Contrast Sensitivity Function (CSF), which maps the sensitivity of the HVS to the contrast perceived, and a correlation exists between the spatial frequency and the threshold once a contrast change is noticed. The sensitivity is defined as the minimal change required to result in a response $S = \frac{1}{C}$, with contrast C being

$$C = \frac{L_{\max} - L_{\min}}{L_{\max} + L_{\min}} \tag{2.2}$$

where $L_{\max}$ is the maximal and $L_{\min}$ the minimal luminance value of the processed waveform [Dal93; War00].

This CSF is subject to the spatial frequency used, and it has a high falloff at low as well as high frequencies [War00]. Such a spatial CSF, for example, is created with the help of a sinusoidally varying pattern presented to spectators, which make a mark where they notice a change in contrast.

As stated earlier, a segregation of the signal is performed in the eye by the ganglion-cells. The signal is interpreted as differences in luminance between red and green as well as between blue and yellow colors. Each of them needs to be modelled with a CSF. An example spatial CSF is shown in figure 2.9.

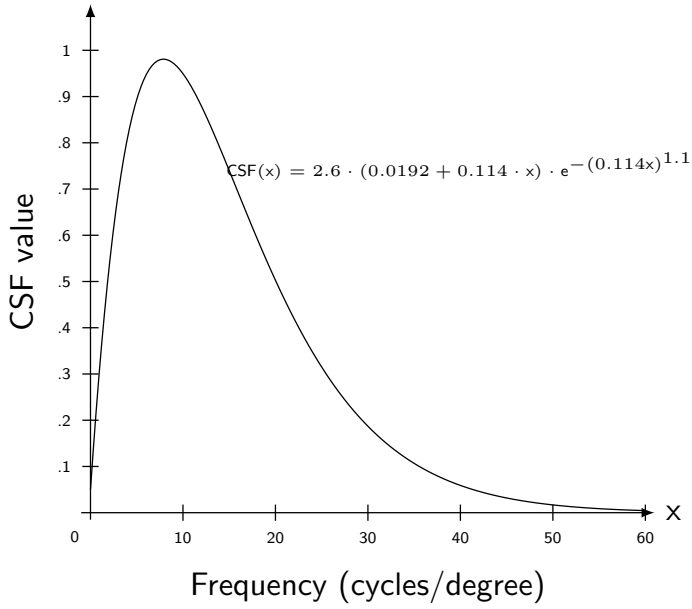


Figure 2.9: A spatial CSF encoding sensitivities relative to the frequency. The frequency is given in cycles per degree. Formula for plotting is defined by [MS74]

The remaining sensitivities, namely light level and signal content, are accounted for with scaling and filtering. Daly states that the sensitivity to the light level arises mainly through "the light-adaptive properties of the retina" [Dal93], and a nonlinear scaling reduces the incoming signal amplitude accordingly.

The signal content is represented with a set of hierarchy filters. Results of physiologic and psychophysical studies indicate that early stages of visual processing are correctly reproduced if applying individual, independent filters to achromatic and chromatic signals [Dal93].

Calculate Differences

In a computer representation, the input signals are compared with a minimal threshold value. The perceptibility of a difference between two input signals is "assessed by applying a CSF in the frequency domain" [Le +06]. By inverting the CSF, the threshold at the current spatial frequency is given. A difference will be visible by a human spectator if the intensity is above the threshold [Pat+98]. Because of this inversion, researchers tend to use CSF functions that can be inverted fast.

The application of CSFs is the main idea behind Daly's *Visual Differences Predictor*. It simulates the HVS with a set of filters to create the three separated layers and tests for perceptibility with the CSFs. The *Visual Differences Predictor* yields a difference map based on two input images, which encodes locations where perceivable differences are present. The approaches presented by Daly [Dal93] and Le Meur et al. [Le +06] apply CSFs on each signal of the visual input, namely an achromatic and two color signals.

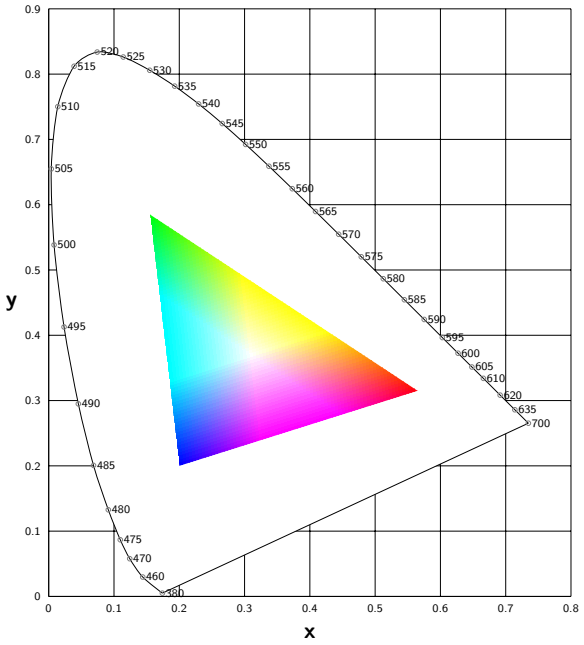


Figure 2.10: The XYZ 1931 color space chromaticity diagram. Note that the displayed colors within the triangle (the gamut) are specific for any display or printing device. As opposed to other visualizations, here the outer colors are not shown, as they would be wrong. Adapted from code of `http://www.fho-emden.de/~hoffmann/pstutor22112002.pdf`, last visited 25.10.2011

2.3.2 Color Spaces

As an alternative to CSF, a popular method is to calculate differences between two images with distance metrics defined in color spaces. A color space defines all representable colors and how they are related to each other.

Well suited for comparison of colors are CIE L*a*b* and CIE Luv color spaces. Both are based on the XYZ color space, which can be visualized in form of a chromaticity diagram (see figure 2.10). Unlike regular color spaces, i.e. RGB, CMYK or HSV, both the L*a*b* and Luv color space account for the non linear response of humans. These correct the MacAdam ellipses in the XYZ color space, so that they become circles [Gla95]. In both, the difference between two given colors is calculated with the ΔE distance metric. Over the years, the definition of the function has been refined several times. The interested reader may be referred to the work of Sharma et al. [SWD05] for a more detailed introduction and definition of the metric.

The fact that there exist two standards arises from the industry, i.e. the paint industry has created these standards independently [War00].

2.3.3 Preattentive Features and Saliency

Another operation performed by the HVS is the detection of important objects in an image or scene. This detection allows to fast select or focus on parts, which may be of interest to a human spectator.

Form	
• Line orientation	• Curvature
• Line length	• Spatial grouping
• Line width	• Added marks
• Line collinearity	• Numerosity
• Size	
Color	
• Hue	• Intensity/Luminance
Motion	
• Flicker	• Direction of Motion
Spatial Position	
• 2d position	• Stereoscopic depth
• Convex/concave shape	

Table 2.1: A list of preattentive features as identified by Ware [War00]. The features are separated into form, color, motion and spatial position.

To simulate this behavior, the same process as in the HVS has to be modelled. This starts with a color difference calculation in the retina and ends at the V1 system where the selection of the important regions is performed. Still, no high level vision participates in these processes, but the information acquired here is useful for the higher cognitive operations.

This approach is inspired by the psycho-physical notion “saliency”. Saliency identifies how an objects figuratively “pops out” of its surrounding and draws the attention of an observer. Koch and Ullman [KU85] give the following definition:

> “Saliency at a given location is determined primarily by how different this location is from its surround in color, orientation, motion, depth, etc." [KU85]

An alternate notion is preattentive processing, which implies that these regions are detected before conscious attention or processing, e.g. during the higher cognitive processes, is performed [War00].

The saliency map, introduced by Koch and Ullman [KU85], contains objects that differ most and collects these in a topological map. The HVS, especially the early vision, will focus on these by placing the fovea upon them. To put it the other way, the saccades of our eyes, i.e. rapid placement of the fovea, will occur preferably on these salient objects. Itti [Itt05] states that visually conspicuous parts within an image often have this ability.

Feature Integration Theory

The visual salience model is based on the *Feature Integration Theory* of Treisman and Gelade [TG80]. It states that visual features are processed in parallel over the complete visual field, and it allows us to perform a selection based only on these features. Due to the large amount of processed data, the HVS exploits this selection capabilities to reduce computation effort or to avoid overload. Such regions where these features occur and poke out relative to others are said to have a high visual conspicuity.

Ware identifies a set of features that are processed preattentively and are organized in categories [War00]. The list of the defined features is shown in table 2.1.

The locations of high visual conspicuity are encoded in a 2d map and are provided to the focus of attention, which then performs the selection on interesting regions in the world [Itt05].

Saliency Map Generation

The architecture (see figure 2.11) presented by Itti et al. [IKN98] processes images by utilizing several image operators and filters. First, the input image is separated into a group of feature images where preattentive extractions are applied. There color-differences, i.e. red-green, blue-yellow and luminance – which is called intensity by the authors –, is extracted and an orientation calculation using a Gabor filter is performed. The first two parts correspond to a CSF and separate achromatic and chromatic signals of the input signal [Dal93]. A Gabor filter, represents a rough estimate of the orientation sensitivity of the HVS and is applied for angles of 0, 45, 90 and 135 degrees.

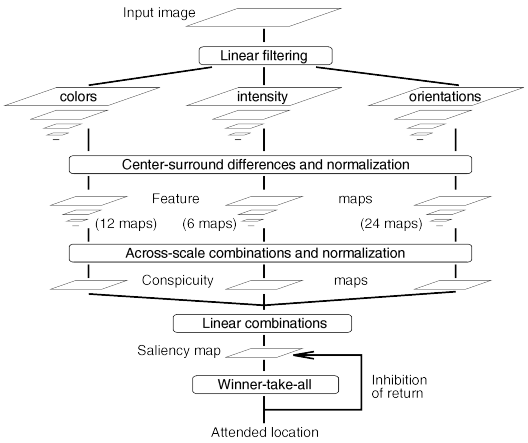


Figure 2.11: The framework to calculate a saliency map as proposed by Itti et al. [IKN98]. The input image is separated into multiple feature maps that are processed. These processed maps are combined into a saliency map. A neuronal network selects the most important areas. This includes an inhibition of return mechanism.

The Gabor function is well suited to model the receptive field properties of neurons in the visual cortex, and it simulates their behavior by performing a segmentation. The function consists of two components: a Gaussian envelope and a cosine wave. The cosine wave defines the direction the filter is being applied identified by the provided angle. This accounts for the alignment of the neurons in the V1 while the Gaussian envelope simulates the excitation and inhibition of these.

As only one angle can be processed at a time, multiple instances of a filter have to be applied to correctly simulate the behavior of the neurons. In equation (2.3) a formula for defining a Gabor filter is presented [Are+05].

$$G(x,y,\theta,f_0) = \exp\left\{-\frac{1}{2}\left(\frac{x_\theta^2}{\sigma_x^2}+\frac{y_\theta^2}{\sigma_y^2}\right)\right\}\cdot\cos(2\pi f_0 x_\theta) \tag{2.3}$$

where x_θ and y_θ are the points being rotated in respect to the angle θ. σ_x and σ_y are the standard deviations of the Gaussian along the x_θ and y_θ axes.

The architecture includes four of the features identified by Koch and Ullman [KU85], but can be expanded with various other features as well. These may include depth, motion relevant features, such as structure-from-motion, or flicker. Also, there is no limitation regarding static images, and so the

model can be adapted for moving scenes or scenes containing motion. Depending on the features that need to be accounted for, an own extraction model has to be provided and needs to be integrated into the existing architecture.

In the next step, the inhibition of the receptive fields is simulated by applying a DOG model. The differences of colors are calculated based on a *color double-opponent* model. It states that in the center of the receptive field the neurons are excited by one color while they are inhibited by the other color. The inverse is true for the surrounding of the receptive field. This way the color differences are given by the following formulas [IKN98]:

$$\mathbf{RG}(c,s) = |(R(c) - G(c)) \ominus (G(s) - R(s))| \tag{2.4}$$

$$\mathbf{BY}(c,s) = |(B(c) - Y(c)) \ominus (Y(s) - B(s))| \tag{2.5}$$

In both equations (2.4) and (2.5) the parameters c and s denote the *center* and the *surround* of the receptive field. $\ominus$ is "the across-scale difference between two maps" [IKN98]. These equations account for the color opponent model of the ganglion cells (see 'Ganglion-Cells' and figure 2.5). For the orientation feature, a set of images with local orientation contrast is created. The resulting images, after application of the operators, are called feature maps as they represent the visible preattentive features.

Normalization of Preattentive Features

A crucial step in the architecture is a normalization operator, which promotes feature maps that have a small number of important features, i.e. large values. These maps identify the conspicuous locations in the input image regarding the applied filter mechanism. The proposed normalization performs the following steps:

1. Normalize values to a fixed range in $[0, M]$.
2. Find the global maximum M, and compute the average of all other local maximas $\bar{m}$.
3. Multiply each pixel with $(M - \bar{m})^2$.

The normalization operator allows to combine the individual feature maps to create an overall saliency or conspicuity map. Itti et al. state that the separate channels compete strongly for saliency while the different modalities are added independently to the saliency map [IKN98]. The result serves as an indicator where the focus of attention will be drawn, and the saliency map provides a lookup by selecting the most salient location, i.e. the largest value, in the map.

The selection of the important regions is performed with an winner-takes-all approach, i.e. the largest value is selected first. Then, an *inhibition of return* is applied that reduces the values in an area around the largest saliency value. So, the influence of other regions in the winner-takes-all selection is increased [IKN98; Hab+01].

Nothdurft [Not00] tested and verified that the individual features do not contribute equally to the final saliency value. This arises also due to the dependency of the strength that an individual feature has in the context [War00]. The relative weight between available features is additionally influenced by task-dependencies, top-down processes or training [Itt05]. But, as the weights are independent of the extraction mechanism, and the final saliency map is created based on the weighted feature maps, a saliency map is adapted accordingly to the varying weights.

Limitations of Preattentive Features

Preattentive processing is limited to simple operations and has strong limitations. As an example may serve the conjunction search of two features, e.g. a search for a shape and a color at once, which is generally not possible by the HVS in a preattentive manner. This is independent of the fact that each feature is preattentively processed [Itt05; War00].

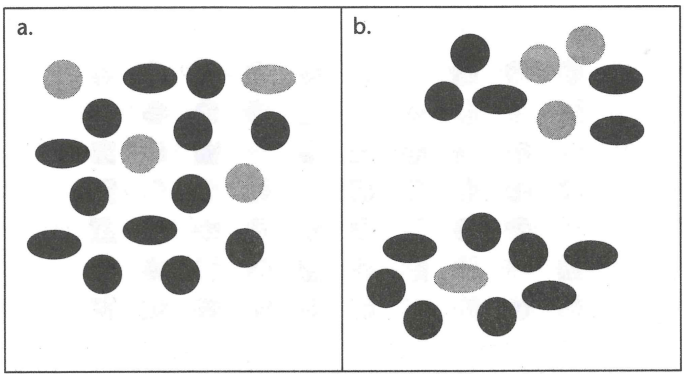


Figure 2.12: Grouping of objects allows preattentive processing and thus a conjunctive search. In the left, no preprocessing is possible while the spatial grouping in the right image allows the HVS to perform a conjunctive search. Image taken from [War00, p. 170].

There are, however, exceptions, and if spatial dimension and either shape or color are to be searched, preattentive processing is possible because the independent groups are scanned by the HVS (see figure 2.12). Once the group with the object to search for has been found, the object is processed preattentively. Ware further references other possibilities to create conjunctions, such as stereoscopic depth, convexity, color, and motion. Yet, they are exceptions, and in general can the selection or conjunction search not be performed in a preattentive manner.

An additional drawback is that a spectator is able to force himself to focus on certain objects or regions. This cannot be accounted for with the saliency model. A high-level vision model would be required to correctly account for such a scenario.

Validity of Preattentive Processing

Itti et al. state that the calculation, the extraction, and the selection based on the saliency map is biologically inspired and neuronal-plausible [IKN98; KU85] because it is a direct representation of the processing performed in the HVS.

The LGN consists of cells with receptive fields that are selective for a certain pattern. These cells only react on their *tuned* input pattern and much less, or not at all, to others. Both V1 and V2 contain cells that are well *tuned* for orientation, size, color (in separated form), stereoscopic depth and motion [War00]. As a topographical relation between the retinal field and the V1 cells exists, the

preattentive processing models the behavior of the HVS in a plausible manner [War00]. Furthermore, excitatory connections in V1 appear to enhance orientation-selective neurons, especially if the input signal belongs to a contour [Itt05]. This validates the individually defined features.

After the HVS has extracted the preattentive features, it focuses locations where most excitation of the neurons has occurred. These are identified by the topological map. Thus, the saliency method is able to identify regions of interest, but is restricted by early vision.

2.4 Level of Detail

The basic idea behind any LOD algorithm is to exploit perception and display limitations. When moving an object further away from the camera, more and more pixels will start to overlap because they are projected onto the same location. The resulting color information will be reduced to the frontmost pixel-color value when no blending of the pixels is performed. This results in a large waste of computation power as this information has to be recalculated for each frame. By reducing the detail of the object, e.g. by removing primitives, the overdraw and computational effort is reduced. However, the removal of primitives has to be performed carefully to retain the quality of the rendered scene.

Thinking of a maximum number of vertices that can be drawn by a specific hardware at interactive frame rates, the LOD allows to display more vertices in near or as important marked objects. Thus, the detail of these objects is increased while detail of unimportant objects is decreased.

The important aspects of all LOD systems are: a) the creation of different LODs for an object, b) a way of storing level specific data in an object and enable access during rendering and c) the selection of distinct levels and transition between the discrete LODs provided.

All aspects together allow us to adapt the rendered scene based on distance or perceptual information. This information is generated only by the computer and does not require any interaction from the user. If the system has been configured correct, the user should not even notice a change in detail.

2.4.1 Creation of Different Levels of Detail

Given a representation of an object, independent of the representation type, a level for the lod is created by either computer algorithms or is manually selected by a designer. The first often lack the ability to account for important features or depend on expensive calculations to allow additional information, e.g. perceptual ones, to influence the levels to be created. The second, however, is quite time consuming as the vertices for removal have to be selected by hand.

Most common are methods that combine both approaches and pregenerate a set of different levels. These are presented to the designer who is able to add or remove details supported by the LOD-algorithm. These details are selected and identified by the designer and thus are dependent on a subjective measure. In computer science, mathematically inspired methods are applied to create these distinct levels and to automatize these process.

Levels are usually created in an iterative manner where each level is based on the previous. This results in a hierarchy of levels as each level serves as a source of the subsequent level, in the regular case a linear chain. The degree of reduction is given by the ratio between the primitives available in two subsequent levels, and it does not need to be constant throughout the object.

Almost all reduction methods generate new levels by removing vertices from the current representation. Hoppe [Hop96] defined a dynamic method that collapses or expands single edges in a representation to generate or reduce detail. The so called *Progressive Meshes* are based on two operations, i.e. *edge collapse* and *vertex split*. To change the appearance of the surface description, which is given by the vertices and the attached edges, either of these operations is applied.

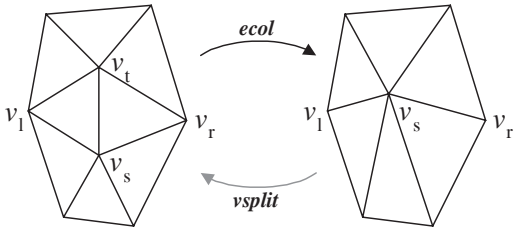


Figure 2.13: The *Progressive Meshes* operations presented by Hoppe [Hop96]. The *edge collapse*-operation removes an edge and merges the two associated vertices. This effectively reduces the mesh size by one. The *vertex split*-operation is the inverse operation, splitting a vertex into two vertices and displacing them. Additionally, a new edge is created between both.

Based on the length of the edge, a selection is performed, and short edges are removed and the respective vertices are merged. This operation is only performed if certain constrains are fulfilled, so that no crossing edges are introduced. The operation remains invertible as long as the information of the edge and vertex position has been stored. In figure 2.13 the results of both operations are visualized. The *edge collapse* joins two vertices while the *vertex split* creates a new vertex and displaces both the old and the new vertex accordingly.

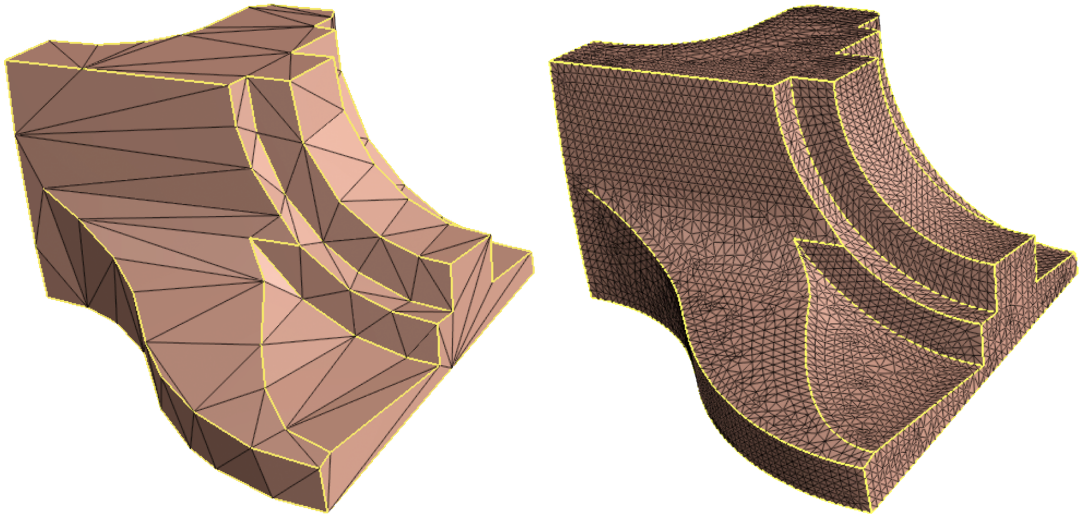
For selection of vertices to collapse, Hoppe applies an energy minimization approach where a mesh M is reduced by evaluating the following function:

$$E(M) = \sum_i d^2(x_i, M) + \sum_{\{j,k\} \in M} \kappa \|v_j - v_k\|^2 \tag{2.6}$$

The equation consists of two parts, a distance measure and a spring simulation. The first sum is the squared distances of all vertices in the mesh to the current vertex x_i while the second sum simulates a spring with rest-length zero and spring constant κ, which guides the minimization process.

A priority queue sorts all operators according to their change to the energy function, and a sequence of operators is extracted that have the lowest energy costs. As an additional requirement, this set of *edge collapse*-operations must only contain legal transformations, e.g. those who do not introduce crossing edges. After application of a single reduction, the energy function has to be reevaluated, making this method very ineffective. However, when generating the levels offline, the resulting LODs can be made accessible during online rendering. [WP01; Hop96].

The application of the reduction allows high compression rates and yields good results. In figure 2.14 a comparison of a low and high detailed results is given. In the low (in figure 2.14a) detailed version the corner regions are preserved because a modified version of the energy function (2.6) has been used (see [Hop96]).



(a) Low detailed version with 200 faces (b) High detailed version with 12.946 faces

Figure 2.14: Comparison of the reduction achieved by the *Progressive Meshes* approach.

Other methods, like QSlim [GH97], are based on a quadric-error metric to select vertices to be removed from the current representation. The new vertex positions are evaluated with the error metric, therefore minimizing the error that is introduced. As a result, a compressed representation is extracted while retaining the overall shape of the object.

The error metric is usually defined to take position or other geometric features, e.g. normal or curvature, into account. The curvature represents inner-object changes, which are salient. Thus, they are more important for the HVS (refer to section 'Visual Perception and the Human Visual System' and chapter 'Bidirectional Saliency Weight Distribution Function' on page 99). For this reason, this has been applied by Lee et al. [LVJ05] to incorporate saliency information into a mesh-based representation. The curvature is calculated with an approximation method presented by Taubin [Tau95]. Taubin's method approximates the curvature of a mathematical surface description, which is based on a local neighborhood extracted from the vertices and edges. The method estimates the principal curvature e.g. the direction of the most changing component. For more information regarding the evaluation of the surface, refer to section 'Calculation of Geometric Properties'.

Utilizing the visual information that the HVS provides, Cheng and Boulanger [CB05] present an automatic LOD-selection based on just noticeable differences. Based on Weber's law, an extension is made for 3d textured meshes that allows a selection of important vertices. The authors state that an increase of texture detail will introduce greater gain than increasing detail as long as the change from one to another level is lower than a predefined threshold.

When encoding the surface positions as spherical harmonic functions [MCA07], an implicit representation of the data is created. Then, only the function needs to be stored, and with it, the mesh representation can be generated. This representation is altered to contain different LODs if the function is reconstructed with varying sampling rate. Yet, this method cannot be used as an online algorithm

due to the large computational cost implied from the spherical harmonics computation.

Level of Detail for Point-based Rendering

For point- and volume-based representations, the locality is crucial for most LOD methods because the point-size has to be adjusted, and no implicit neighborhood information is available. In the QSplat rendering system proposed by Rusinkiewicz and Levoy [RL00], a hierarchy is created on top of the input polygon set. Based on these polygons, the input points and the initial point-sizes are extracted as well as other necessary surface information, e.g. the surface normal. The input triangles are sorted by the edge length with a brute force method, and unimportant surfels are removed from the basic data set. A space partition algorithm is utilized to create a kd-tree with the remaining input surfels. The inner nodes are placed on the mean positions of their children and their size is set, so that it encloses all child surfels. This way, a bounding sphere representation of the input data is generated. By traversing the hierarchy and providing an abort criterion, the displayed LOD is influenced. As the generation of LODs in the hierarchy only uses position information, it can only be assured that the surface is represented accurately if the object is displayed using the leaf nodes.

In the case of *Progressive Splat Decimators* presented by Wu et al. [WZK05], a sequence of applied merge operators (see figure 2.15) is recorded, and a coarse base splat set is generated. Similar to the *Progressive Meshes* method from Hoppe, the operators are inverted to result in a set of detail operators. Starting with the base splat set, the object is refined or coarsened dynamically using the operators available for each splat. This way, the final representation of the object is extracted. Again, the LOD-refinement is influenced by an abort criterion.

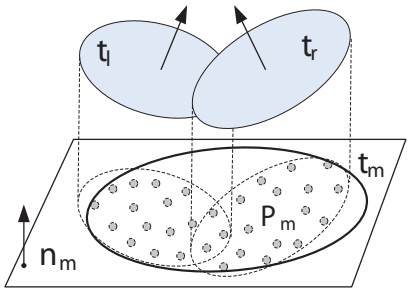


Figure 2.15: The merge operator applied to the fine input point set as presented by Wu et al. [WZK05].

The point-based representation comes with an additional requirement for the LOD-hierarchy: A node must represent the aggregate attributes of all the points in the sub-tree. Otherwise, the display will produce artifacts when rendering different levels. Usually, the position is created from a mean position – median values are possible as well –, but for point-sizes the maximum value has to be provided instead [GP07]. This way, a node represents an approximation of the whole sub-tree. When traversing towards the leaves, no holes are introduced as well.

Another approach is to encode the change in the surface using normal displacement [Gus+00; PGK02]. This way, only the difference in normals needs to be stored. Once this displacement has been identified, a more detailed representation of the surfel, or vertex, is generated by displacement, i.e. moving the surfel along the normal by a given factor. This is an application of the displacement mapping technique [Coo84], which leverages a height field to move the position of a vertex along the surface normal's direction. The proposed techniques, however, solve the inverse problem as the LOD depends on calculating the displacement values. As stated by Guskov et al. [Gus+00], the normal

displacement is stored efficient with wavelets, which offer a large compression ratios.

2.4.2 Generation of Details

Progressive methods allow to introduce new surfels, leading to a more detailed object than the plain data could provide. These newly introduced surfels are displaced, so that certain properties are fulfilled, such as a regular sampling of the surface. These methods, however, share the problem that a further refinement does not necessarily mean that the representation will be more accurate.

Given a neighborhood representation, e.g. the one-ring neighborhood, an interpolation of the surface is given by defining a cubic Bézier-triangle. This triangle is defined for each triangle in the fan defined by the one-ring. Vlachos et al. use a cubic Bézier-triangle and positioned the new vertex in the center of gravity, i.e. the barycenter while Guennebaud improved this method by displacing the boundary points of the triangle to avoid a flat area in the reconstructed surface [GP07]. The sampling needed to define whether a surfel is inserted or not. This is controlled by evaluation of the newly created neighborhood. The refinement is aborted once a maximal value in the triangle mesh is smaller than a given threshold. In this case, an insertion of new surfels would not lead to an improvement of the surface representation.

When using Bézier-patches, both a smooth representation and a low polygon variant of any surface can be generated [WP01]. The advantage is that the mathematical definition of the surface allows to take samples of individual positions at run-time. The representation of a curved surface can be smoothened if necessary. A major disadvantage is that discontinuities cannot be modelled easily and may require more complex functions, such as Non-Uniform Rational B-Splines (NURBS). Another mathematical description, usable for refining as well as for coarsening the representation, is a Moving Least Squares (MLS) approach. This is commonly applied to point-based representations and allows to minimize an input set, which was correctly reconstructed with the MLS-operators [Ale+03]. The *Point-Set-Surfaces* allow interpolation of the described surface with varying degree of detail. But, again some extensions are needed to include discontinuities in the smooth representation, e.g. edges or gaps. This can be accounted for with different algebraic base objects and has been proposed by Guennebaud and Gross [GG07].

Another option is a LOD-refinement based on fractals, which basically is representable with progressive methods. Fractals were defined by Mandelbrot [WP01] and describe the self-similarity of an object. Using a production rule, a more detailed representation is retrieved by applying this rule to the current representation. Applications for fractals include the generation of landscapes, objects or even cities [Gre+03]. The latter can be constructed with a procedural generation technique as well. Both, fractals and procedural generation, allow the dynamic generation of surface information based on a set of rules. This also applies for L-systems and mathematical representations as well (see Oberguggenberger and Ostermann [OO11] for a survey).

Procedural generation techniques often require a random generator to create a seed that defines an initial representation. As long as the system behaves deterministic, the objects generated with the

same seed will be equivalent. Furthermore, the definition of such systems is far from being intuitive, and changing a single value in the mathematical description can lead to a completely different result. Yet, it is not possible to represent all types of objects using procedural formulas, which is one of the major reasons why this type is not very popular in games or commercial products.

Since the introduction of shader model 5 presented with DirectX^{TM}11, it is possible to evaluate a displacement directly on the graphics card. The hull and domain shader control the positioning of the newly generated vertices. The hardware internally defines a Bézier-patch based on four neighboring vertices and emits more vertices for rendering. One application is to displace these newly generated vertices, resulting in a more detailed representation of the object. This partially removes the need to generate a full LOD-hierarchy in advance, but has to be reevaluated each frame.

2.4.3 Storing the Levels of Detail

As different resolutions are stored for each level of the object, these representations are referred to as multi-resolution representations. Various methods exist to store the data in a file for later access, and these mainly depend on the method applied for detail reduction.

The simplest method is to store the individual levels in a sequential manner. The levels can also be distributed among multiple files, but both do not yield a satisfying results since a large amount of data is required.

A more complex method is to make use of spatial data structures like octrees [Sam84] or kd-trees [Sam94; Sam95]. These are ideal for point-based and voxel-based representations. In those cases, the data is compressed efficiently, and an out-of-core data layout can be made available without a large performance penalty. The tree data structures also matches the base representation of the multi-resolution generation for the point-based methods. An octree offers a higher compression rate than a kd-tree representation because the positions of the inner nodes are encoded with fixed relations between the parent and its children. Thus, the position, assuming a root node position and extend have been defined externally, is encoded in less than 3 bits per node [BSW08]. The octree representation, however, may be unbalanced if the surface cannot be split regularly into the eight octants [GP07]. The kd-tree leverages this problem by dynamically selecting a split axis along with a split value. By selecting the median of the current set regarding the split axis, the set is divided equally into two halves, and so less empty cells are generated.

In contrast to the spatial partitioning methods presented before, local vicinity is encoded in a bounding volume hierarchies (BVH). These are common in animations, i.e. the I-COLLIDE collision detection presented by Cohen et al. [Coh+95], as they allow to detect and resolve collisions between two objects efficiently. The bounding volumes do not require to be completely spatially sorted, e.g. inner nodes may change their relation to the parent without changing the parent's volume. When the surface changes, only those nodes are updated that have their assigned bounding volume changed, e.g. due to movement. Bounding volumes are also well suited for LOD-representations and allow quantization of the internal data [RL00].

For methods like *Progressive Meshes* and *Progressive Splat Decimators*, it would suffice to store a minimal representation and provide the expansion rules. When loading an object, the representation is expanded on-the-fly to the desired LOD. The operators are further reduced with common compression methods as some similar expansions may exist [WP01]. The problem with these methods arise in an efficient memory layout and lookup of the operators. Also, the selection or expansion may be expensive, and precious rendering time would be wasted. But, they allow the current representation to be refined and altered without any a priory knowledge of the final object. This allows a streaming data transfer, which is typically found in a server-client setup. The client receives the initial data set consisting of the minimal object representation. The operators are sent in correct order and allow an in-place expansion, avoiding expensive transfer of object data. If the locally generated data is cached, the generated surfels can be retrieved fast when they are needed again. In [GP07] a "Least Recently Used" strategy is proposed where outdated, old surfels are discarded first when creating a finer representation of the object.

2.4.4 Transitions between Given Levels of Detail

The different LODs allow to show the object with the desired degree of detail, but only switching the level yields no visual improvement. The change in the level will be perceived as a flicker, which is a very strong saliency feature. Thus, it will be probably noticed by a human spectator.

With flicker, we mean in this context the discontinuity of pixels between two consecutive frames. When the LOD of an object is changed, the projected pixels values will differ because the source data has been altered. To avoid these effects, the frequency of the performed change must be adapted or smoothened. In terms of a CSF, a value may not be above the visible threshold in respect to the displayed version of the last frame. This means that the displayed version cannot be changed suddenly without introducing visible artifacts. These artifacts are based on the temporal discontinuity between two subsequent frames and should be avoided.

The second method of a transition between the individual levels is to change the levels when it is no longer perceivable. This might not introduce visible artifacts, but works against the main paradigm of a LOD-system. A change should be performed as early as possible to reduce the geometric detail in the scene. Only then an acceleration of the rendering is achieved in general. A common approach is to display the two different versions simultaneously and perform a time-based blending between both, slowly converging towards the newly introduced level. In the beginning, an overhead is introduced by showing both LODs-versions, but after blending has been completed, the lower detailed version is displayed without generating flicker.

If a progressive representation is available, the changed vertex positions can be interpolated over multiple frames. This results in smooth change as the individual vertices are moved according to the detail operator provided. But, this change also creates the impression of an organic surface, which is not desirable for solid objects [GW07].

Giegl and Wimmer [GW07] lessened the mentioned problems by imposing the requirement that

one object level must be fully opaque before the other one may be faded out. They state that their operation is to be interpreted as a constructive solid geometry (CSG) LOD-approach. The transition starts at LOD level 1, blends towards the CSG union and then finally blends over to level 2. They also propose a method to avoid z-fighting artifacts by writing only one object in the depth buffer.

2.5 Calculation of Geometric Properties

Depending on the provided 3d data, surface properties, such as point-size, surface normal, or curvature, need to be calculated. These properties are either required by the rendering system or perceptual evaluation, e.g. when computing preattentive features (refer to 'Preattentive Features and Saliency'). The 3d data ranges from position up to rendering specific information, such as texture coordinates or surface colors. In 3d applications, for example, a surface normal is usually required for illumination or culling methods.

For all calculations, a local frame, here the neighborhood of the evaluated point, is required to define a smooth surface representation based on the discrete sample positions given. The definition of a neighborhood in 3d may be as simple as traversing connecting edges, e.g. in a mesh representation, but may also be very complex if only position information is available. The latter case is encountered when creating 3d objects from laser scanning data. Usually, the geometric properties need to be extracted before rendering can be performed [Hop+92; PG01]. Although methods exist that overcome this restriction, for example *Instant Points* presented by Wimmer and Scheiblauer [WS06], higher quality rendering and perceptual evaluation require the presence of surface properties.

Surface properties are covered in the field of differential geometry, which provides methods to calculate and define different kinds of surfaces. To create a surface from sampled data, topological information in form of a neighborhood is required.

2.5.1 Neighborhood Extraction

The neighborhood of primitives, e.g. a vertex, is a set of vertices that is ordered by a certain, usually euclidean or geodesic, distance metric. The euclidean distance metric defines the shortest distance from one position to another in space while the geodesic gives the shortest distance between two positions along the surface of an object. In our case of the surface properties extraction, the calculation of all neighbors, also known as all nearest neighbors (ANN), is not required, and a subset, the so called k-nearest neighbors (kNN), suffices to define a smooth surface representation. The size of the neighborhood k directly influences the smoothness of the surface. The value of k depends on multiple factors, e.g. the sampling rate of the data provided or the target application.

The complexity of defining a neighborhood varies with the input data, and the simplest case is given with a mesh-based representation. In this case the one-ring neighborhood, i.e. all neighbors with an direct edge to a vertex, is given by all incident edges. More effort, however, is needed for point-based data as neighbors are not implicitly given by this representation.

The naïve way is to search k-nearest neighbors with a brute-force algorithm by calculating the distance of each vertex to all other vertices in the data set. This method has a time complexity of $O(n^2)$ with n being the number of vertices in the data, and thus it is not practicable for large data sets. It is possible to improve this naïve method by reusing calculation results. This enhancement increases the memory consumption of the algorithm, but does not change the average time complexity. Despite the fact that the k-nearest neighbor search can be performed in a preprocess, a faster method would be appreciated.

Accelerating Search with Hierarchies

One way to solve this problem is to create a search structure like an octree or kd-tree based on the input data. The data is partitioned in the space along distinct axes with a fixed or free split position. The hierarchy is constructed with a recursive algorithm, reducing the input data set during each partition step. The recursion allows efficient parallelization of each sub-branch, which increases performance during creation. The construction of a hierarchy has an time complexity of $O(n \log n)$ because it is an comparison-based sorting algorithm.

A search of k-neighbors has a theoretical time complexity of $O(k \log n)$ for a single point, yielding a complete time complexity of $O(nk \log n)$ [SSV05]. This method performs much better than the brute-force algorithm and exploits local coherency through partitioning. However, a search requires to traverse the complete hierarchy because neighboring cells or nodes do not provide connection information.

An observation is that points with small distances are often found in local nodes, i.e. they have a common parent node. Therefore, an expansion of the same nodes occurs during search. Callahan [Cal93] presented a parallel algorithm that decomposes the input data into multiple sets and connects these based on the bounding boxes of the nodes. The technique is known as *well-separated pair decomposition* and allows the generation of k-nearest neighbors in $O(n \log n + kn)$. Instead of re-traversing the hierarchy, the connection is used to extract the neighbors.

Space Filling Curves

Another way to calculate nearest neighbors is to benefit from a space filling curve, e.g. a z-order curve or *Morton order*. It allows to sort the individual points by a mapping on this curve. The z-order curve has the property to fill the space based on a bit representation. So, multi-dimensional points are mapped to a position on a one-dimensional curve while preserving locality of the input data. This allows to order the points based on the implicit curve definition. In figure 2.16, an example of a z-curve in a 2d scenario is shown.

The z-curve is created by comparing single bits of an integer value. The algorithm in 2.1 extracts the most significant dimension and returns the result of a “less than” ($<$) comparison based on this dimension. The determination of importance is performed with an “exclusive or” (xor) operation. The coordinates of two points for a single dimension are compared with the xor operation, and the

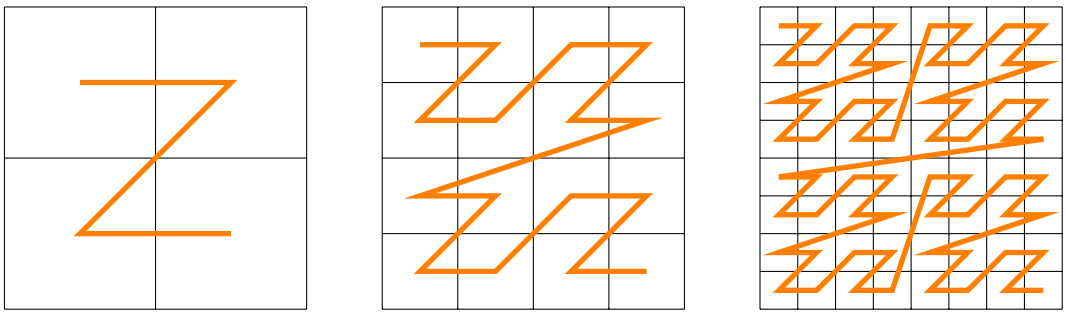


Figure 2.16: A Z-Curve and the corresponding cells. As the curve is defined in this case for 2 dimensions, a quad-tree is mapped to the cells. For a 3d scenario the cells map to an octree. The figure shows three iterations of the z-order curve fitting the provided cells.

```
bool compare(point p, point q)
{
    double x=0,y=0;
    double dim = 0;
    for(int i=0; i<3; i++) {
        y= xorMSB(p[i],q[i]);
        if(x<y) {
            //save dimension
            x = y;
            dim = i;
        }
    }
    return p[dim] < q[dim];
}
```

Listing 2.1: Comparison operation for creation of the Morton order. Each dimension, here in total 3, are checked for the most significant bit, and the largest dimension is then compared. Code derived from [CK10].

one with the most significant bit is stored. Due to the bit representation, the curve was initially defined for integer data, but recently an extension to floating point values has been presented by Connor and Kumar [CK10]. They propose to convert both floating point values into a normalized representation in the form $m \cdot 2^e$, which is applicable to all floating point values. If the exponents e are equal, the mantissa m are compared with the xor operation (see `xor_mantissa` in listing 2.2). Otherwise, the larger exponent is used. Due to the additional computations required, it performs slower than the integer version. A pseudocode of the algorithm for extraction of the "Most Significant Difference Bit" is shown in listing 2.2.

In the case of three dimensions, the z-order curve describes all positions of an octree in a well defined order. The nearest neighbors are extracted by selecting a point on the curve and following it along the curve in a certain distance. The distance from the source point on the curve directly correlates with the distance in 3d space. When performing a kNN for all points, a so called Nearest Neighbor Graph (NNG) is created.

In [CK10] a parallel implementation of a nearest neighbor search with a z-order curve has been

```
int xor_mantissa(int a, int b)
{
 return a^b|(unsigned int) 0.5;
}

int xorMSB(double a, double b)
{
 int x = exponent(a);
 int y = exponent(b);
 if(x == y) {
  int m_a = mantissa(a);
  int m_b = mantissa(b);
  int z = xor_mantissa(m_a,m_b);
  x = x-z;
  return x;
 }
 if(y < x)
  return x;
 else
  return y;
}
```

Listing 2.2: XOR-operations on the floating point values as proposed by [CK10]. First of all, the exponents are compared followed by the mantissa.

presented. Most steps in the search allow parallelization including the sorting and selection of nodes. The presented algorithm first sorts the input points by the z-order curve with normal parallel sort routine. Afterwards, a prediction of the nearest neighbors is performed. This is done with a sliding window along the curve that selects $2k+1$ points as potential nearest neighbor candidates for each input point in parallel. The authors state that an additional refinement step is required as the nearest neighbor can also be located in other neighboring cells. Points assigned to a cell at (x,y) may have also neighbors in all surrounding cells, e.g. $(x+1,y)$. But, this is only the case if the corners of the bounding box of the current solution is greater than the already searched range [CK10]. A parallel recursive algorithm is applied to refine the current solution until all neighbors have been correctly identified.

It is possible to omit the refinement step to increase performance, but in our case, a higher quality is desired. This creates more stable results than an approximate solution. Despite the fact that the algorithm is conceptional more complex, the time needed for creating the search structure remains $O(n \log n)$, due to the comparison-based sorting algorithm. The selection of the neighborhood, however, reduces to $O(nk)$ for a single processor. If a parallel model is applied, the authors presume a constrained CREW PRAM with p threads, the time complexity is $O(\left\lceil \frac{n}{p} \right\rceil k \log k)$, including the parallel sorting step [CK10]. This is slower than the theoretical optimal solution of $O(n \log n + nk)$, but does not have large constants, which renders this algorithm more useful. In tests, the presented method performs faster than other methods by a factor of three [CK10].

2.5.2 Curvature Calculation

The HVS is sensitive to various features, one being the change of a surface. This change is described by surface normals because they are the first derivative of the surface positions. These indicate how the surfels are aligned and oriented. A change of these normals, i.e. the second derivative, provides essential information of the underlying surface. This information is important for many techniques in rendering and surface analysis [Rus04].

In differential geometry, the properties of a surfel are defined in a local coordinate system that origins in the surfel's position. With a surface normal $\vec{n}$, a tangent vector $\vec{t}$ and a bi-normal $\vec{b}$ the so called tangent-space is defined. In figure 2.17, a visualization of the tangent space origin around a surfel is shown.

Various types of curvature of a surface exist, the most important ones are principal, mean, and Gaussian curvature. The principal curvatures represent both maximal and minimal curvature values regarding the local tangent frame of a surfel. The Gaussian curvature is the product of the principal curvatures while the mean curvature is defined as the sum of both divided by two. Thus it yields an average value.

To calculate curvature values, a mathematical representation of the surface is required. As in our case only sampled data is present, implicit representations that allow an exact curvature calculation are excluded. Therefore, the surface needs to be approximated. Interpolated or estimated curvature values have to suffice.

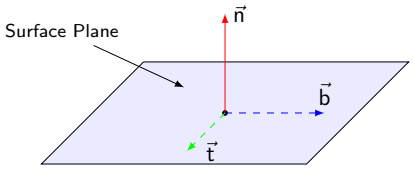


Figure 2.17: The tangent space defined around a point on the surface. It describes the surface, the bi-normal, and tangent vectors that define an orthonormal basis. One application in computer graphics is bump mapping.

Several researchers have addressed the calculation of curvature values. Assuming a mesh-based representation, an extraction and selection of a nearest neighborhood is performed directly on the data, and no preprocessing is required. The method presented by Taubin [Tau95] approximates the curvature in the one-ring neighborhood given by the incident edges of a vertex. The algorithm calculates the tensor of curvature of a surface and results in principal curvatures and directions after computing a closed form of the eigenvectors and eigenvalues. Rusinkiewicz [Rus04] presented a calculation method for curvature of irregular meshes. It is based on a finite-differences approach, and it has fewer outlier estimates as well as an increased accuracy in comparison to other existing methods. Recently, Griffin et al. [Gri+11] published an online algorithm that performs curvature calculation on the GPU for arbitrary triangular meshes. The algorithm includes normal, area calculation and performs a curvature tensor extraction to calculate principal directions. Additionally, the derivative of the principal curvatures can also be determined.

By partially describing a smooth curve or surface, the curvature is extracted more accurately. As this method is also based on sampled data, an approximation occurs during the reconstruction of the surface. Good results are achieved with a MLS-surface, which allows various curvature calculation methods. In the base representation of a MLS-surface, the eigenvalues and eigenvectors of a sur-

```
float getNormalVariation(Vector3 normal
                         , Vector3 *neighbor_normals
                         , int neighbor_count)
{
    float dot = 0.0f;
    for(int i=0; i < neighbor_count; i++) {
        dot += DotProduct(normal,neighbor_normal[i]);
    }
    dot /= neighbor;
    return clamp(1-dot,0.0f,1.0f);
}
```

Listing 2.3: Normal variation in a local neighborhood. It calculates a rough approximation of the curvature at the given surface point. A dot-product is calculated to account for the angles, which are summed up. The result is averaged and clamped.

fel may be extracted, but this requires some lengthy computations. An improved method has been presented by Yang and Qian [YQ07], extracting the curvature of a point set surface directly. The authors present closed formulas that result in the curvature of a surfel on a MLS-surface. The MLS-representation, however, requires neighboring input points to create the interpolation surface. These neighbors are extracted either by a mesh-based representation or by a window filter, which is applied to point sets. The filter rejects points with a distance to the center point greater than a given threshold value.

In this thesis, we use an solution that approximates the curvature, called normal variation [SK10]. The surface normal is compared to the normals of its neighbors, and the variation is averaged (see pseudocode in listing 2.3). This results in an approximation of the curvature as the change of the normal is calculated. If curvature is calculated in a preprocess, e.g. if the object is static, a more accurate method should be used instead. We use this method to process models online. Furthermore, it is possible to extract object boundaries with this method by interpreting the dot-product to 1 when an object boundary is found. This is the case when a normal is compared with an invalid value, e.g.$(0,0,0)^T$.

3. State of the Art

In this chapter, current systems and techniques are presented that are either used within this work or provide the starting point for novel research. As with the background (see chapter 'Background' on page 5), first rendering algorithms will be presented, followed by current perception-based rendering methods. At the end of each section, a conclusion is given evaluating each part. This will lead to the definitions of the theses presented in chapter 'Theses' on page 63.

3.1 Mesh- and Voxel-based Rendering Methods

Newest methods and algorithms operating on mesh-based representations are commonly introduced along with the next generation of a shader model. In the current model, shader model 5, the generation of object details is transferred to the GPU with the tessellation shader. New versions of graphics APIs, such as DirectX^{TM}or openGL, usually promote features made available by the next generation of a shader model.

In the following subsections, most recent advances and techniques in mesh- and voxel-based rendering are presented. These include mathematical representations as well as GPU accelerations, which are achieved by transferring processing onto the graphics card. Some techniques perform calculations that overcome the main application of the graphics card, i.e. the projection of a scene. These are referred to as GPGPU programs and perform universal calculations with the GPU.

3.1.1 Mesh-based Rendering

Mesh-based approaches draw much of their success from the easy representation of surfaces. By connecting single vertices with edges, a face is defined and a flat surface is created. This reduces the amount of primitives needed to represent large areas.

Yet, this advantage is also a disadvantage as large effort is required to achieve smooth surfaces. Most objects in real life do not have sharp edges and thus have to be represented using a large amount of primitives. By evaluating a mathematical model to describe a smooth surface, primitives are generated and stored in the object during a preprocess.

This increases the size of the object and has a large impact on the performance. Also, the creator has to control the final result by hand. Several tools are available to support a designer during this step, e.g. by providing control of a smooth surface with Bézier- or NURBS[1]-curves or by reduction tools that allow removal of unnecessary primitives (refer to section 'Level of Detail' on page 26). But, this approach is tedious, and a static representation of an object is created. Recent advances, however,

[1] Non Uniform Rational B-Splines

tend to transfer this preprocess step to the graphics card. These smooth surfaces are evaluated on the graphics card and allow to dynamically adapt the presented object.

First, an introduction to current mathematical models, which allow to create smooth surfaces, is given, before algorithms are presented that generate details directly on the graphics card.

3.1.2 Smooth Surfaces Using Curves

To describe an object in more detail, different methods are available. Displacement methods create details by inserting new vertices along a predefined displacement vector and value. This is similar to the progressive LOD-methods presented in section 'Level of Detail' on page 26.

An alternative method for creating smooth surfaces is to use a Bézier-curve, or more precisely Bézier-patches or Bézier-surfaces in the 3d case. The description of smooth 2d curves has been discovered by De Casteljau, but the mathematical representation has been published by Bézier.

A Bézier-patch is mathematically defined as a function based on a set of $(n+1)(m+1)$ control points $\mathbf{P}$. The patch warps a unit square defined in the k-dimensional space. In this case, k depends on the dimensionality of the used control points. When applying 3d control points, a 3d representation of a Bézier-patch is created.

A Bézier- or bi-quadratic-patch is parametrized by the position $\mathbf{Q}$, which is given with coordinates (u,v) on the patch [WP01]:

$$\mathbf{Q}(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2} P_{ij} B_i(u) B_j(v) \tag{3.1}$$

where B_i and B_j are Bernstein polynomials. These are defined as

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i} \tag{3.2}$$

For the bi-quadratic patches, which are restricted to 3d, these Bernstein polynomials are:

- $B_0(u) = (1-u)^2$
- $B_1(u) = 2(1-u)u$
- $B_2(u) = u^2$

Bézier-patches allow to create smooth representations without the need to sample and store discrete positions because they are evaluated fast on nowadays hardware. In combination with displacement methods, a dynamic LOD with sharp details in the object is created. A designer only needs to define the displacement to control the behavior of the surface.

However, Bézier-curves and patches have several drawbacks, such as the non-localness or the relationship between degree of control points and their surface points [Rog01]. These drawbacks can be

circumvented by using B-Splines. These define a segment over multiple control points, which influences the underlying patch. Yet, a comparison of Bézier-curves and B-Spline-curves is quite misleading. As opposed to Bézier-patches, B-Splines describe a piecewise, composite curve or patch [WP01].

Unlike Bézier-patches, B-Splines extend the property of being smooth due to the second derivative continuity in position (C^2). Any combination of B-Spline basis functions will have this property because it is a property originating from the basis functions themselves. A curve or patch is given by the following formula:

$$\mathbf{Q}(u) = \sum_{i=0}^{m} \mathbf{P}_i B_i(u) \tag{3.3}$$

These basis functions are split into three groups. The first group are the uniform basis functions where each function is a translated version of each other. The second include non-uniform basis functions and are freed from this restriction. Bézier-curves and patches fall into this group as well. The last group consists of non-uniform rational basis functions. For example, a B-Spline surface can be defined as follows:

$$\mathbf{Q}(u,w) = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} B_{i,j} N_{i,k}(u) M_{j,l}(w) \tag{3.4}$$

In contrast to Bézier-patches, B-Splines and especially NURBS may contain any degree of continuity. Therefore, NURBS-curves or patches are often found in modern CAD applications because sharp features, which are desirable for many design situations and exist on real life objects as well [Rog01], are representable with them.

NURBS consist of control points with according weights and a knot vector. This knot vector defines a sequence that determines where and how the control points are influenced. The convex hull – also referred to as the control net – surrounding the patch limits the surface. It is possible to interactively place and move the control points. Also, the knots and the weights of each control point can be changed [WP01]. The positions on the surface are calculated with four-dimensional polygonal control vertices ($B^h_{i,j}$ with h being the homogeneous weighting factor) and the non rational B-Spline basis functions $N_{i,k}(u)$ and $M_{j,l}(v)$ [Rog01]:

$$\mathbf{Q}(u,w) = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} B^h_{i,j} N_{i,k}(u) M_{j,l}(w) \tag{3.5}$$

where $N_{i,k}(u)$ and $M_{j,l}(w)$ are the B-Spline basis functions in the biparametric u and w directions. $B_{i,j}$ are the vertices of the polygonal control net.

Rational B-Splines, including the non-uniform ones, have certain properties, which have been covered by Rogers [Rog01]. The most important for the context of rendering are presented here:

- A rational B-Spline patch of order k,l is C^{k-2}, C^{l-2} continuous.
- A rational B-Spline patch is projection invariant, meaning any projection applied to the control

net results in the projected version of the surface.

- If triangulated, the control net forms a planar approximation of the rational B-Spline patch.
- The surface lies within the convex hull of the control net.

Different methods exist to evaluate any smooth patch, but all require many computations. Ranging from a naïve to an iterative solution, the complexity of the algorithm varies. The iterative solution has a constant time complexity as long as the basis functions remain constant, e.g. the set of basis functions is not changed or altered [Rog01].

3.1.3 Evaluation of Curves Using the GPU

With the addition of hardware tessellation (shader model 5), it is now possible to change the existing methods of creating different LODs by incorporating specialized shaders. Unlike the generative process made available by the geometry shader introduced with shader model 4, the tessellation patches only reside inside the specialized shaders of the graphics card memory.

The renderpipeline (refer to figure 2.1) is extended with new stages, i.e. a *Hull Shader*, a fixed function *Tessellator* and a *Domain Shader*. The first and the last stages are programmable to control how the new patches are generated [Loo+09].

The *Hull Shader* defines the convex hull the patch is being placed in, like Bézier patches. As input a list of indices referencing the control points is provided, which limits the generated patch. The *Tessellator* uses this input to create a sampling pattern, and the number of samples or, more precisely, the vertices to be emitted are defined for each patch individually. The *Domain Shader*, which is called by the *Tessellator* during generation, evaluates the patch data and modifies it if necessary. In combination with the input convex hull and the barycentric coordinates, which are provided for each vertex in the patch, the final vertex position is returned to the *Tessellator*. The connectivity information defined by the input then allows to create the final patch [Loo+09].

Tessellation Schemes

Multiple methods exist to control the tessellation for a given patch. Especially linear interpolations, local construction schemes or approximations of the Catmull-Clark Scheme (CSS) are well suited for the application in the tessellation stage of the graphics cards. These schemes are independent of external information, e.g. the mesh structure, which is important for a shader implementation.

If a linear interpolation of a mesh is applied, the *Hull Shader* is not required and is implemented as a passthru. The *Domain Shader* performs a barycentric or bilinear interpolation. This has successfully been applied to terrain or environmental objects [Bon11].

While linear interpolations create repetitive patterns easily, local construction schemes operate on a single input primitive and refine it. For example, the *Phong Tessellation* presented by Boubekeur and Alexa [BA08] refines a quadratic geometry patch and introduces new points in it along with Phong shading properties, i.e. smooth surface normals.

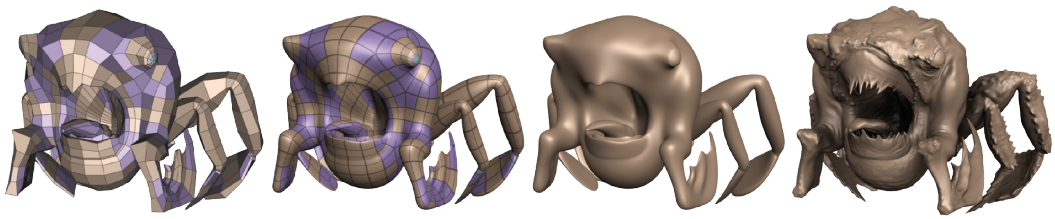


Figure 3.1: Results of the Gregory patch-based tessellation. The presented method directly utilizes the tessellation shader.

The CSS defines a subdivision rule for arbitrary topology. It is a generalization of a bi-cubic uniform B-Spline surface and is applied to a polyhedron. A recursive application of rules evolve this polyhedron to the limit surface defined by the B-Spline. The CSS has become a standard for free form shapes in movies and computer games [Loo+09].

For applications within a shader, approximative solutions are better suited because of the complex calculations required to evaluate B-Splines. An acceleration is achieved when using the GPU, and an implementation has been presented by Loop and Schaefer [LS08]. The same authors refined their approximation with Gregory-patches and provided a solution on the GPU [Loo+09]. Some results of the proposed method are shown in figure 3.1.

Gregory-patches are a modified tensor product along with triangular polynomial patches. They assure that a set of patches meet with tangent plane continuity and are a rational extension of Bézier-patches. They share many common properties, for example, the boundaries are Bézier-curves and the end-points are interpolated [Loo+09]. To create a Gregory-patch, the one-ring neighborhood and the face is needed.

The approximation with Gregory-patches in hardware starts with the hull shader and a primitive with three or four vertices. Based on these, the shader extracts a set of special vertices, which are required for the approximation. It is notable that each of these control points is a linear combination of the vertices in the one-ring neighborhood. If all patches share the same connectivity, the weights have an unique set of values [Loo+09].

The domain shader evaluates the individual vertex positions, which are emitted by the tessellation shader. The evaluation, however, is not performed in the Gregory form because the rational nature of these patches require expensive calculations. Instead, a transformation into polynomial Bézier form is performed and the positions are evaluated with the algorithm of De Casteljau.

A drawback of this approximation scheme is the incorrectness of the normals because the Gregory-patches and their corresponding functions have non-trivial derivatives. Loop et al. state that the approximation of the normal is exact “along the boundaries of the patch and continuous over the interior” [Loo+09]. Another disadvantage is the large number of control points required to tessellate the surface, e.g. for a quadrilateral 20 and for a triangle 15 control points for each individual Gregory-patch are required.

Generation of Details with Tessellated Surfaces

The question, which algorithm to use, cannot be answered easily since a trade-off between quality and speed exists. Local construction schemes are easier to implement and require less calculations, which improves performance, but lack high visual quality. Interpolation or approximation schemes provide higher detail or better control of the surface appearance, but require complex computations.

Independent of the selected refinement algorithm, structure and texture increase the plausibility of an object to be shown. The combination of fine grained patches and displacement mapping creates highly detailed surfaces, and it overcomes restrictions of illumination-based approaches. For example, methods like bump-, normal- or relief-mapping only simulate the appearance of a structured surface whereas displacement mapping alters the representation of the object and so generates both real cavities and bumps.

Displacement mapping not only allows correct illumination, but also avoids artifacts, such as loss of depth impression at low angles, which is common in bump-mapping techniques. The technique has gained importance with the introduction of the tessellation shader because highly tessellated surfaces, i.e. a large amount of primitives, are required to create a smooth displacement.

Usually positions of individual vertices are transformed along the normal with the following formula [Mun+10]:

$$d(u,v) = p(u,v) + \vec{n}(u,v)t(u,v) \tag{3.6}$$

where p is the point on the tangent frame. $\vec{n}(u,v)$ denotes the surface normal while $t(u,v)$ determines the displacement to be applied to the point.

Munkberg et al. improved the normal method by introducing conservative bounding boxes based on displaced Bézier-patches. In combination with culling methods, these offer large performance improvements. An early culling of a bounding box avoids generation of unneeded details, which otherwise would be discarded by the z-buffer in the pixel shader.

3.1.4 Alternatives to Tessellation and Curves

Besides high processing power of today's graphics cards, other methods than geometric detail are available to achieve high quality images or to remove vertices in an object without loosing detail.

To create photo-realistic impressions, texturing methods are commonly used. During the mapping of an image onto the surface, a filtering of the image has to be applied to reduce sampling artifacts. These artifacts arise through the perspective projection of the surfaces, and usually linear or bilinear filters are applied to safely reduce the image size, i.e. the sampling theorem is not violated.

Modern graphic cards provide additional filter methods and especially anisotropic filters increase the quality of texturing. The filtering is performed along different directions in space and is supported by current hardware. This method does not require large amount of processing time, but it yields an even sharper texture mapping. The application of anisotropic filters is not limited to texturing, and

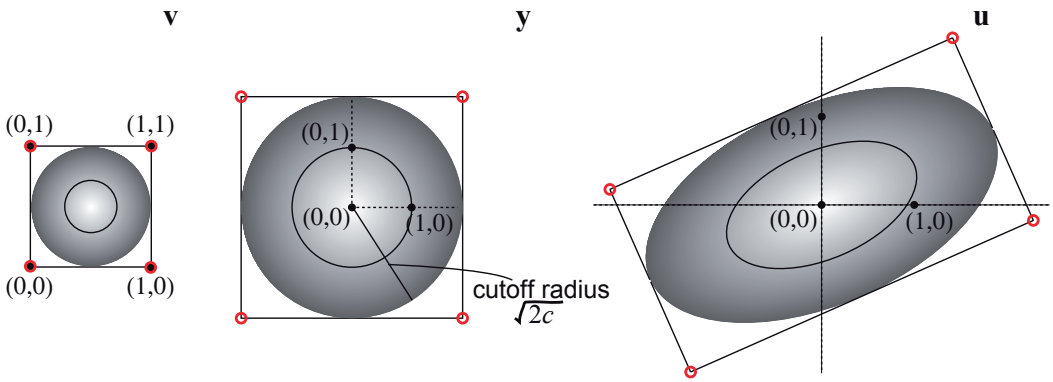


Figure 3.2: Derivation of the EWA resampling filter from a texture-mapped polygon. Image taken from [RPZ02].

reconstruction methods also benefit from them, too. The Elliptical Weighted Average (EWA), for example, is an anisotropic filter and results in fewer sampling artifacts than a uniform filter [GP07]. The anisotropic property of the EWA arises through the projection of a surfel into screen-space, and the derivation of a warped EWA resampling filter is shown in figure 3.2.

To create the impression of structure on a plain surface, bump- and normal-mapping techniques simulate this behavior without the need to generate vertices along with a displacement. By warping the surface normal in the illumination calculation, the impression of a changing surface is created. However, this approximation becomes more and more ill-suited at small elevation angles, and the missing structure will become obvious.

This issue has been addressed with the relief-mapping technique presented by Policarpo et al. [POC05]. It performs a local raytracing of the surface depending on the visual angle. The underlying relief is usually represented with a height-map that encodes bumps and cavities. Due to the raytracing, self shadowing effects are covered as well, and thus a better impression of the simulated structure is created. However, a search of intersection points needs to be performed for each pixel of the surface. In some situations visual artifacts occur, especially if the search fails to find intersections due to sampling artifacts.

3.1.5 Voxel-based Rendering

Similar to the mesh-based rendering techniques presented before, modern methods accelerate voxel-based rendering with the help of the processing power of today's GPUs. In contrast to surface representations, such as mesh- and point-based rendering, voxels allow to simulate volumetric effects, for example, dampening of fog and refraction in gaseous materials.

In the GigaVoxels algorithm, a hierarchical octree is utilized to reduce the spatial data information [Cra+09]. An octree representation encodes position information with less than three bit per node and so provides good compression [BSW08]. Additionally, LODs directly arise without further mod-

eling, which defines a so called brick-map [GP07] (see figure 3.3). A brick-map is a 3d mipmap representation, which is interpreted as an adaptive octree. In an adaptive octree, the subdivisions for each level is controlled by the number of voxels present within each block or brick. Each brick approximates a part of the original volume in the octree hierarchy and is stored in a so called *brick poll* in the GPU memory. Rendering subsystemss usually provide 3d textures for direct memory access. However, memory of graphics card is limited. Especially when volume-based objects are used, the required space exceeds the available storage. Thus, only some parts of the complete data is placed in the memory, and a selection strategy needs to be applied.

Crassin et al. [Cra+09] utilize a “Least Recently Used” (the authors use the term “last recently used") cache strategy to mark voxels that have not been used and may be removed from the graphics card memory. During rendering, nodes are marked with the current timestamp, and if a node is requested to be refined, the oldest node is replaced with a new brick. Only during this replace operation data needs to be transferred to the graphics card. A feedback methods identifies the nodes that need to be replaced. A selection mask contains a bit-vector with all voxels to keep, and a compressed node index texture is extracted from this mask. With this texture, the stored voxel positions are updated, and the algorithm resumes with rendering.

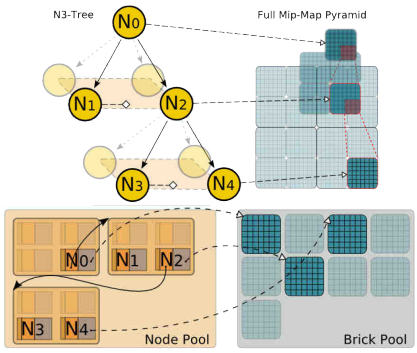


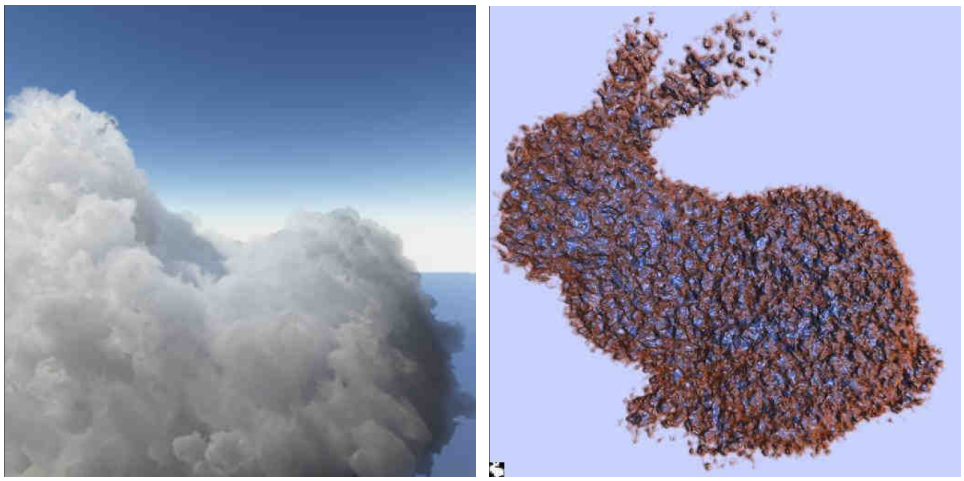
Figure 3.3: The data structure used by the GigaVoxels algorithm presented by Crassin et al. [Cra+09].

For display, a marching algorithm accumulates the colors and opacity values along the view rays. An iterative descent method starts from the tree root and traverses the tree. In a first step, the origins of the view rays are located within the tree structure, and the descent stops at a predefined level. The ray is then followed node by node. If a so called constant node has been found, the value stored in the node is integrated. These constant nodes are either empty or have a fixed color in case of a leaf.

The selection for the marching algorithm is optimized by taking advantage of the octree structure. Depending on the current depth, the indices of the child nodes are determined with the formula $\text{int}(x \cdot N)$ where N belongs to a N^3 tree. In such a tree, each node is subdivided into N^3 children[2]. This fast retrieval of child nodes increases performance and builds the key to a GPU implementation.

In figure 3.4, some results rendered using the proposed algorithm are shown. All screenshots are taken from a prototype application, which runs at interactive frame rates. It is notable that volumetric objects, i.e. clouds, appear extraordinarily realistic because a smooth accumulation along the view ray is performed by the ray-marching algorithm.

[2] An octree is N=2 since each node is divisible into $N^3 = 2^3 = 8$ children.



(a) Clouds (b) Hyper-textured bunny

Figure 3.4: Results generated with the GigaVoxels algorithm. Note the high quality of the rendered clouds. Individual bricks are identified with a mask and replaced at run-time. Screenshots taken from [Cra+09].

3.2 Point-based Rendering Methods

Point-based rendering methods have advanced from their plain representations in the mid 1980s to more elaborated methods. Still, point-based rendering is the most straightforward method of creating images from 3d data and modern implementations allow smoother and more visually pleasing results.

With the introduction of QSplat rendering [RL00] and the definition of surfels by Pfister et al. [Pfi+00], point-based rendering once again gained in importance. Since then, research has constantly resulted in novel methods and even led to combined representations of mesh and point-based representations, the so called hybrid models [CN01].

Most important achievements in rendering regarding point-based methods are due to smooth surface representations. In contrast to flat shading models available in the mid 1980s, a Phong model – despite its lack of physical correctness – is used as a working basis for rendering nowadays. In the following, two methods are presented, which are required to generate smoothly illuminated images with point-based representations.

3.2.1 Elliptical Weighted Average Splatting

In computer graphics and especially 3d rendering contexts, the generation of smooth shaded surfaces is one of the most basic tasks. As representations are usually undersampled, i.e. there are holes in the surface, an interpolation of vertices during the rasterization stage is required to generate missing samples, i.e. pixels. Here, splatting algorithms start to introduce large errors in form of holes or hard illumination edges due to the lack of neighboring information. As no topology information is available, no interpolation between neighbors can be performed using the standard rendering methods

available on the graphics card or rendering subsystem.

One method to solve this problem is the EWA splatting as proposed by Zwicker et al. [Zwi+02]. It allows smooth interpolation of neighboring surfels with the help of a multi-pass approach. Instead of squares or circles, ellipses are generated during rasterization because they have less aliasing effects. Zwicker et al. state that the splatting approach is to be understood as a resampling operation in terms of signal processing. The discrete representation of the surfel allows to establish a mapping between the surfel and its position in pixel-space. Any resampling process needs to obey the Nyquist limit to avoid aliasing and reconstruction errors[3]. As a discrete signal is given, some basic signal processing methods are required.

The solution is to apply a filter kernel shaped like an ellipse – hence the name – when creating the image, which accumulates signals in the neighborhood. This reconstruction or resampling is performed during rendering and Gross enumerates four steps [GP07]:

1. Reconstruction of a continuous signal from the discrete input samples. A weighted sum of samples c_i and a reconstruction window r_i with source coordinates t integrate neighboring information and create a continuous signal.

$$f(t) = \sum_i c_i r_i(t) \tag{3.7}$$

   In an implementation, usually a 2d Gaussian reconstruction kernel defined on the tangent plane of the surfel is used.

2. The input function is transferred into target domain with a mapping operator m.

$$f'(x) = (f \circ m^{-1})(x) = f(m^{-1}(x)) \tag{3.8}$$

   Despite the given definition, the tangent plane is not constant for each splat and an implementation needs to provide an individual mapping function.

3. A low pass filter h is applied to the mapped function, which results in the anti-aliased footprints of the splat. The notion resampling kernel is also common.

$$\bar{f}'(x) = (f' \otimes h)(x) \tag{3.9}$$

4. The output samples are generated by an impulse train i

$$p = \bar{f}'(x) i(x) \tag{3.10}$$

   The implementation uses the z-buffer algorithm to accumulate only visible footprints in the affected pixel locations. This process is also known as footprint rasterization.

[3] For real photos this, however, can never be assured.

A major drawback is that no direct implementation on a GPU and no smooth illumination is possible because only color information is blended. Instead, neighbor information would be required. Different approaches remove these limitations of the CPU approach through hierarchical methods [BWK02]. But, an implementation that utilizes all GPU capabilities leads to a more efficient processing due to vector-based hardware available on GPUs.

3.2.2 Phong Splatting

With the Phong splatting approach the EWA splatting was transferred to the GPU, and also "Perspective Accurate Splatting" was included. It was proposed by Botsch et al. and extended to a deferred rendering by the same authors [BSK04; Bot+05]. The deferred rendering approach reduces time complexity and is specially suited for scenes with large numbers of light sources.

GPUs do not support the rendering of EWA primitives, and thus no straightforward implementation of a EWA splatting algorithm is possible. However, with the introduction of programmable shaders, a GPU is able to perform the required operations while the overhead is kept to a minimum. The algorithm proposed by Botsch et al. is manually optimized to reduce the instruction count and minimize the clock cycles spent for each program on the GPU.

Along with well optimized vertex and pixel shaders, multiple passes, e.g. drawing the scene multiple times, are needed to generate a final rendered object. The result is further enhanced by employing a perspective correct splatting method, which acquires information from the view frustum and viewport. With the near and far planes n and f and the camera adjusted depth value z', the depth buffer entry for a pixel is corrected with the following equation [GP07]:

$$\mathrm{z}(x,y) = \frac{1}{z'} \cdot \frac{fn}{f-n} + \frac{f}{f+n} \tag{3.11}$$

The proposed deferred algorithm performs the following passes during each frame to generate the projection of a point-based objects:

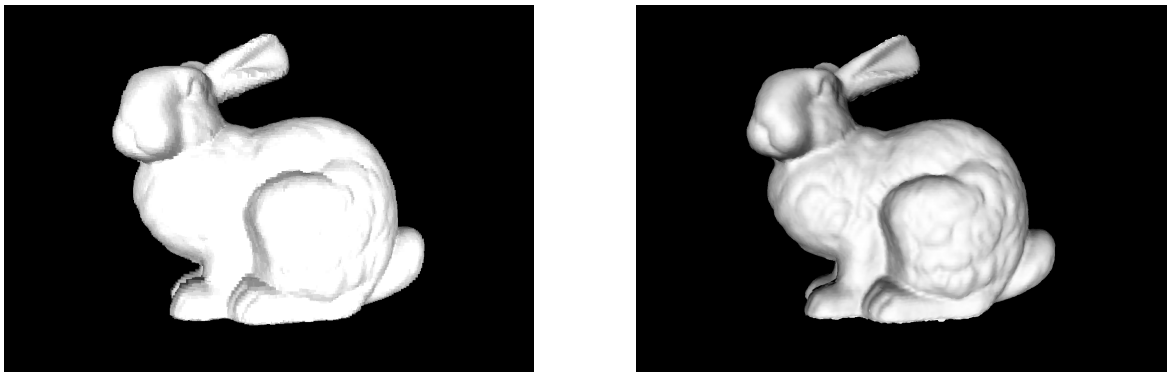
1. The depth of all surfels are splatted perspective correct into a depth buffer. This depth value is adjusted with an additional small delta value.

2. The attributes of the object are splatted reusing the previously created depth values. This assures that only the frontmost surfels are drawn and rasterized while others are rejected by the depth buffer. Valid attributes are scaled with an elliptical weight and accumulated into a framebuffer.

3. In a full screen quad[4], the attributes are normalized by dividing each surfel by its accumulated weight. This results in an interpolated value defined by neighboring points.

As an accumulation of overlapping splats is performed (see 2nd pass), the early-z culling method is used to remove hidden surfels from the scene. In the first render pass very simple vertex and pixel

[4]This is a quad that fills the entire screen.



(a) Plain rendering (b) Phong splatting

Figure 3.5: Comparison of the plain rendering of surfels against the Phong splatting approach. Figure 3.5a shows the result of the Stanford bunny with rectangles as primitives. 3.5b is the same object, but with Phong splatting.

shaders are applied to allow fast evaluation. Once the depth values have been calculated, the buffer is used for early rejection to avoid overdrawing of individual splats.

The depth values are transformed with a small value during the first pass, which yields a so called fuzzy-z buffer. This fuzziness is needed to create a smooth interpolation of surface features, i.e. color and surface normal, because a single splat does not provide the required information alone. Only the frontmost surfels should be provided when performing the interpolation. This avoids artifacts from invalid surfels, e.g. ones that are part of another surface or are hidden by the frontmost surface.

The reconstruction filter defined by the EWA method controls to what extend neighboring information is accumulated and leads to a smooth representation of the surface. Instead of an interpolation of color information, i.e. a Gouraud shading, the surface normals are used, i.e. Phong shading. The normals of the neighbors are accumulated, resulting in an un-normalized normal vector at the interpolated pixel. No special requirements are defined for the interpolation and thus may be applied to any attribute.

The final screen-space pass normalizes the individual splatted attributes. The deferred rendering approach enables to apply additional effects and illumination without re-rendering the scene. The illumination then only requires local, pixel-based information. An advantage of applying illumination in screen-space is that only visible surfels are accounted for and unnecessary calculations are avoided. Some advanced techniques also allow spot lights and point lights in high count while retaining interactive frame rates by defining an extend of these light sources [Lau].

The visual quality of the Phong splatting method surpasses plain rendering approaches and achieves similar results to the EWA approach (see figure 3.5). Additionally, the Phong splatting allows to render the object with the help of the GPU.

3.3 Conclusion of Rendering Methods

The incorporation of hardware anti-aliasing methods, like Phong splatting, and the small size of triangles allow to work without connectivity information. The memory that is freed this way can be used, for example, for storing additional details in the geometric representation.

As point-based methods share the same data layout as voxel approaches, so called out-of-core methods can be use to draw large data sets at interactive frame rates. For example, the method of exchanging memory used in the GigaVoxels approach [Cra+09] may also be applied to point-based rendering techniques. The smooth reconstruction of the surface is useful for voxel rendering techniques, but unlike surface representations, a marching algorithm or other accumulation methods are needed to account for volumetric features.

For both, the main problem arises due to lack of proper hardware support. With current GPGPU applications, this disadvantage will vanish soon and other rendering methods, e.g. raytracing, will become more popular. Adamson and Alexa [AA03] proposed a raytracing method using *Point-Set-Surfaces* where intersections between surfels and view rays need to be evaluated.

Modern graphics hardware allow in-place refinement of mesh-based objects, and a new shader stage has been developed specifically for this purpose. Multiple methods exist ranging from predefined methods to Bézier-patches. As this information is solely generated on the graphics hardware and does not need to be transferred from graphics card, this leads to a massive increase in performance.

Hardware tessellation for point- and voxel-based methods would open new possibilities to control the LOD of an object. However, current control methods for LOD are based purely on mathematical descriptions or predefined textures, and thus it remains questionable whether or not hardware-based tessellation, in this manner, is applicable to perception-based rendering at all. A combination of both the real-time refinement and perceptual information would define a real-time, visual perception-based LOD-algorithm.

3.4 Perception-based Rendering

All object representations and rendering methods share a common drawback – usually no perceptual information is accounted for. However, most of them allow a modification of an object during drawing or by selecting the LOD.

The idea to employ perceptual information during rendering is not new. Yet, the evaluation of perceptual information usually requires expensive computations, and the focus so far was on non-real-time rendering methods. This includes raytracing, pathtracing algorithms, global illumination methods or animation simulations. All proposed methods share that in the time of their publication no hardware acceleration for these was available.

Recent advances in this field of study allow raytracing or global illumination solutions to be generated on GPUs. Purcell et al. [Pur+02] presented a raytracing algorithm, which is directly computed on the graphics card. Based on this method, nVidia™introduced OptiX™, which provides raytracing

on the GPU in form of a library. However, computations are still expensive and acceleration methods that employ visual perception remain hard to achieve.

Researchers presented different methods, all extracting differences in a precalculated, often estimated, scene to evaluate whether an algorithm should continue the refine process, or the difference is no longer perceivable by a human spectator. This dynamic abort allows to save calculation time as no fixed recursion or iteration count is required.

3.4.1 Metric-based Approaches

Ramasubramanian et al. [RPG99] utilize the visible difference predictor from Daly [Dal93] and enhance it with a physical error metric. First, a threshold needs to be defined, which will be used by a rendering algorithm. Based upon a *threshold-vs-intensity* function, a minimum contrast is extracted and passed to the rendering system. The visual system is simulated by processing input images with multiple bandpass mechanisms. A hierarchy, a so called *contrast pyramid*, is created and local differences are extracted. In combination with CSF, a threshold a map is generated, which identifies regions where additional refinement is required. As an example, the authors provide a global illumination approach – using a pathtracing algorithm –, adaptively aborting the recursion based on this threshold map.

Another approach is to incorporate perceptual influences directly into the LOD-creation step. Reddy [Red97] applies a CSF during an offline preprocess and extracts interesting visual features. In combination with size, velocity, and eccentricity, the reduction is performed by primarily omitting details in unimportant regions.

The purely spatial CSF is enhanced by introducing an *Animation Quality Metric* (AQM) presented by Myszkowski [Mys02]. In combination with an image-based rendering approach, a walkthrough of static environments is accelerated. An intermediate frame is interpolated based on the keyframes of the animation. The AQM defines where the keyframes have to be placed to reduce the number of interpolated pixels within each frame. Depending on the error threshold, the number of keyframes is reduced, and the overall animation calculation receives a speedup.

The modeling of perception with a CSF allows to make use of fast filter methods. In combination with a Fast Fourier Transformation (FFT), the filters are applied efficiently to an input signal. The output, however, needs to be transformed back for the remaining calculations. Still, pixel processing is accelerated and thus the overall time needed in an implementation is reduced. Of course, this may be a drawback if the transformation becomes a bottleneck during visual perception calculations.

Drettakis et al. [Dre+07] present a CSF approach purely implemented on the GPU (see figure 3.6). The individual contrast functions are encoded as lookup tables in the graphics card's memory. These lookup tables are chosen to be small, so that memory is saved. In combination with a layered rendering method, a threshold map is generated, which contains tolerable local luminance difference values similar to the approach of Ramasubramanian et al. [RPG99]. These layers and their individual threshold maps are recalculated each frame, exploiting the ability of modern GPUs to create multiple

render targets in one pass. So n-layers are created during one rasterization step. The layer generation is performed in additional render passes, and the results control rendering parameters, e.g. the drawn LOD. When using L layers, a total of $L+L-1+(L-1)*9$ render passes are required to generate all necessary layers with this method. For example, when $L=3$, a total of 23 render passes are performed.

This method requires two issues to be solved. One is how to perform a comparison between the current representation and the object with the largest visible artifacts identified by the layers generated. The other is how to transfer the computed threshold information to the CPU because the data is solely calculated on the GPU.

The first issue is solved by introducing an initialization step, which compares the current representation with the highest detailed representation available to derive an error value. A depth pass rendering is performed, so that only visible pixels of object in question are evaluated. Each layer is compared to the optimal layer utilizing a delay strategy. Only a fraction of the objects assigned to a layer are rendered with high quality while the others are drawn at lowest quality possible. Only the high quality objects are then evaluated and compared. Based on the results of the comparison, one of three possible update methods is performed:

- **Increase** the LOD if the visual error is above a threshold
- **Maintain** the LOD if the visual error is tolerable
- **Decrease** the LOD if too many details are shown (the visual error is below another threshold)

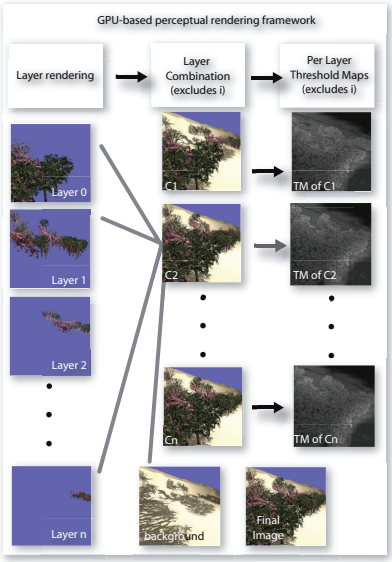


Figure 3.6: The interactive rendering pipeline for perception-based LOD as presented by Drettakis et al. [Dre+07]. First, n layers are rendered and combined in all possible combinations. The results are processed to get the threshold maps, which are used to reduce the LOD of the objects in the scene.

The second issue, the supply of the calculated data to the CPU, is addressed with the so called occlusion query feature. The result of an occlusion query holds the number of pixels that passed both stencil and depth test in the current scene. The authors alter the query to return the number of pixels that are above the defined error threshold. This is achieved by applying a shader, which discards individual pixels in the processing when they are below the threshold. The occlusion query includes only pixels that have an error higher than the threshold provided, and thus an overhead of transferring unnecessary data is avoided.

Another method leverages a progressive rendering algorithm, which is based on a Delaunay triangulation. Farrugia and Peroche [FP04] present a combination with a perception-based metric, and an

evaluation of the results has been performed. The vision model used was defined by Pattanaik et al. [Pat+98], which is the same as that by Ramasubramanian et al. [RPG99], but without the perceptually-based physical error metric mentioned earlier. The distance between two pixel values is defined by the $L^{2.4}$ metric:

$$d(p) = \|p_2 - p_1\|_{2.4} \tag{3.12}$$

where p_1 and p_2 are the individual pixels, which are to be compared.

To accelerate the computation, only a small fraction of estimated scene is taken into account. The principle is similar to other approaches where the resolution is reduced to accelerate computations. Farrugia and Peroche approximate the results in a quad-tree partition. Random samples within each “block" are used to estimate the differences. This block image approximately contains the same information as the full solution [FP04].

3.4.2 2d Saliency-based Approaches

Instead of deriving or defining a metric, the following approaches generate a saliency map, or a similar topological representation, to identify where higher detail would be required. Le Meur et al. [Le +06] refined the model of Itti et al. [IKN98] and verified the results with various eye tracking experiments. Independent of early vision mechanisms (in their work called “passive selection”) humans have to focus on a particular region in the scene. Thus, “[s]accades are therefore a major instrument of the selective visual attention” [Le +06]. Le Meur et al. model the focus in form of a fixation map, which is a saliency map with some enhancements. One is prioritization of the center of the image over peripheral regions. The results are given by chromatic and achromatic CSFs, and the individual features are combined in a DOG model. A masking is applied inspired by the work of Daly [Dal93], and a reinforcement of the individual signals is performed utilizing the surrounding information. This models the center-surround inhibition of receptive fields (see ‘Receptive Fields’ on page 15) and is implemented by convolution of the current signal with a normalized weighting function. The authors state that a Butterfly filter enhances the contours and provides grouping. The filter consists of a directional term D and a proximity term generated from a blurred circle C:

$$B(x,y) = D(x,y) \cdot C_r(x,y) \star G(x,y) \tag{3.13}$$

where

$$D(x,y) = \begin{cases} \cos(\frac{\pi}{2}\frac{\varphi}{\alpha}) & \text{if } -\alpha < \varphi < \alpha \\ 0 & \text{else} \end{cases} \tag{3.14}$$

and $\varphi = \arctan(\frac{x'}{y'})$ and (x', y') are rotated coordinates by angle Θ. With parameter α the opening angle of the Butterfly filter is defined, which influences its angular selectivity. The $\star$-operation denotes the convolution.

The final fixation map is generated by summing the individual channels weighted with a Gaussian around the center of the image. This accounts for the experiment setup where users should look at the center of the screen before an image was shown.

The large computational effort needed to create saliency maps is transferable to modern graphics cards. Xu et al. [Xu+09] present a multi-GPU implementation based on nVidia^{TM}CUDA kernels. The derivation of a saliency map is completely performed on graphics card, and the authors state that speedups of over 8.5 times are achieved in comparison to well tuned CPU implementations. This furthermore means that the saliency calculation is not a bottleneck for online calculation on modern computers.

For the creation of a saliency map, the model of Itti et al. [IKN98] is used. The individual stages of the architecture are converted into a GPU friendly layout. Special treatment was required in the case of the Gabor filter. The convolution with a Gabor filter is costly, and therefore a FFT is performed along with its inverse. Despite this additional step, the complexity is reduced from $O(n^4)$ to $O(n^2 \log n)$ with being n the number of pixels. Given an input image with a resolution of 640x480, the first two mipmap levels are skipped, and a 256x256 image is generated that holds all other sub-images on which the FFT is applied to. The original Gabor filter is re-sized by repetition to match this resolution, and the images and the filter are multiplied[5] once. The reconstruction with the inverse FFT must be performed as often as sub-images are present in the combined image. The performance is further increased by utilizing multiple GPUs, which perform the calculation of input images in parallel, and so the overall performance is increased.

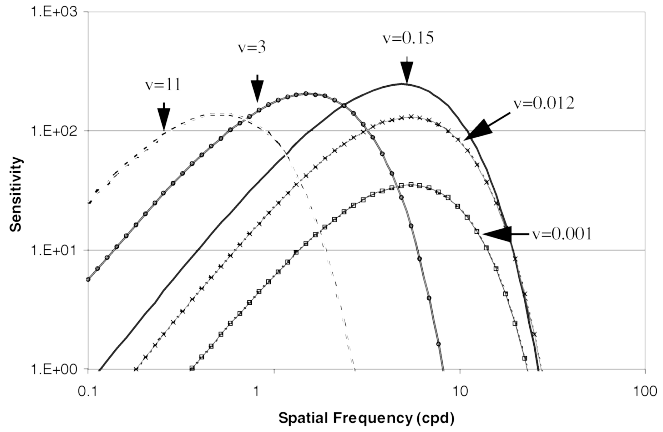
The DOG is separated and approximated with two convolution filters. The first filter models excitation behavior of a receptive field, and the second, larger kernel accounts for inhibition. Both are applied in an iterative normalization step, and so "simulates the local competition between neighboring salient locations"' [Xu+09].

Once the saliency maps have been generated, a rendering system can use these to select important regions or to provide a stop criterion for progressive rendering algorithms. Yee et al. [YPG01] present a framework that accounts for moving objects in a scene. Here, the saliency map definition is enhanced with spatiotemporal information to create a so called Aleph Map. The spatiotemporal sensitivity represents the amount of error that is tolerable, and the saliency map encodes important regions in the scene. If the objects are moving, the spatiotemporal CSF also accounts for the reduced sensitivity of human spectators.

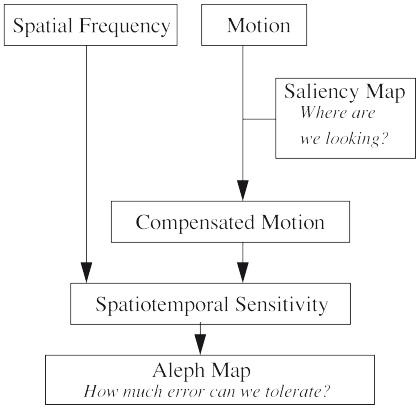
Yee et al. [YPG01] define a CSF to model the spatiotemporal information that depends on the retinal velocity. In figure 3.7a, an example CSF is seen that is influenced by the velocity of the eye. The faster the eye moves the faster the falloff of the sensitivity curve. This spatiotemporal CSF has been derived from the results of Kelly [Kel79] who studied the decrease of sensitivity in dependency of the eye movement.

The authors intentionally ignore the top-down processing in "favor of a more general and auto-

[5] due to the conversion with FFT, it is not a convolution



(a) An exemplary spatiotemporal CSF. The faster the eye is moving, the greater the reduction in sensitivity. Initial test were performed by Kelly [Kel79] and were also utilized by Daly [Dal98]. Figure taken from Yee et al. [YPG01]



(b) The steps performed to acquire an Aleph Map [YPG01]. Motion, saliency, and spatial frequency are included to derive a more fine grained identification where higher detail is required.

Figure 3.7: A spatiotemporal CSF, which is used in the derivation of an Aleph Map. The CSF also depends on the velocity of the object to be tracked. The Aleph map on the right allows a more fine grained identification when moving objects are in the scene.

mated bottom-up approach" [YPG01]. Visual attention is modelled with the architecture presented by Itti et al. [IKN98]. Motion is incorporated in an additional conspicuity channel and is estimated based on the displacement of pixels in consecutive frames (see figure 3.7b).

A purely image-based implementation is possible, but the authors enhance the estimation by adding geometric information available from the 3d representation. Additionally, discretization artifacts occurred in an exemplary openGL implementation, and a raycasting method has been implemented to circumvent this problem.

The Aleph Map is applied to a global illumination method, i.e. an irradiance caching algorithm, which calculates surface color information. The authors measured an acceleration up to a factor of 9 independent of the samples used for rendering.

Haber et al. [Hab+01] enhance the saliency method, which models the bottom-up visual attention, with a task map that enforces selection of important regions. Their top-down process generates a bounding box for each non-diffuse object in the scene. The distance from the center along with its pixel count is extracted, normalized, and the values are combined to form the task map. Along with a saliency map, generated by the method of Itti et al. [IKN98], important regions in the scene are identified. This visual processing controls a raytracer by adapting the number of recursions for each ray and by positioning the samples according to the maximal saliency value.

The authors state that two basic methods for processing the saliency map results are possible. The first is the method proposed by Itti et al. [IKN98], a winner-takes-all approach. The object with the largest value is selected and *corrected*, i.e. its detail is increased until another object has a greater saliency value. This, however, does not account for the possibility that a spectator will change the focus to other regions with also high saliency values. The authors therefore propose an alternative

method, which is proclaimed to yield better results. The most salient object is refined only a few times and its saliency value is reduced by a certain amount. Afterwards, the selection of the most salient object is repeated.

To create an estimate of the scene, a hardware accelerated projection is performed and provided to the visual salience model. A static representation of the scene is used, and a Global illumination is calculated for each object in a preprocess. The results are stored either in a texture or in the vertices as a color value. Therefore, this method is restricted to static scenes.

The authors propose the following layout. A main thread provides the user interface and exchanges scene estimates and perceptual results. Perceptual analysis and raytracing are distributed over several threads. In a loop the following steps are performed in each iteration [Hab+01]:

1. Set the camera and age the samples in the cache.

2. Render the scene using the graphics hardware without lighting because color information is stored either in textures or as a vertex color. Write the stencil buffer with the id of each object.

3. Store the content of framebuffer in a texture (T_{FB}), and clear it by deleting the stencil information.

4. Warp and splat the samples from previous frames with the stencil test. This assures that the splats are only on the surface of each individual object.

5. If time is left, acquire samples from invoked raytracing threads.

6. Blend the current scene with the stored framebuffer T_{FB}.

7. Copy the stencil and framebuffer to the analyzer thread if an update was requested.

The visible objects are refined when they have any non-diffuse components. The global illumination – performed as a preprocess – assures that diffuse parts are evaluated correctly and do not need to be recalculated. The raytracing algorithm calculates specular components of the objects in question. It operates on the same scene description as the graphics hardware including the camera setup. Some results along with the calculated saliency map are shown in figure 3.8.

The precalculated global illumination has the disadvantage that moving objects cannot be accounted for. A change in position would invalidate all preprocessed results and a recalculation would be necessary. However, the method allows a real-time refinement and walkthrough of static scenes. The detail of objects increases when more samples are acquired by the raytracing threads. Also, salient features are prioritized in the scene, which is a result of the combination of bottom-up and top-down modelling. The top-down process, however, does not completely simulate higher cognitive processes because only a selection of the available regions of interest is performed.

(a) Fast scene estimate (b) Calculated saliency map (c) Final result

Figure 3.8: The individual stages of the perceptually guided splatting approach presented by Haber et al. [Hab+01]. The first image shows the result of the hardware accelerated image. The specular objects in the scene are black because no illumination has been applied yet. In figure 3.8b, the saliency map can be seen. In the last figure, the final result is shown.

3.4.3 Methods for Higher Visual Processing

Haber et al. and Yee et al. state that top-down processing is important for modeling visual attention. Yet, both try to reduce its influence to a minimum by either providing a rough approximation without any higher cognitive processes or by simply discarding it. Even without an implementation of high-level vision, a large speedup is achieved by utilizing a low-level vision model.

When a task is provided to the spectator, object detail may be reduced even further. Cater et al. [CCW03] focused on the exploitation of *inattentional blindness*. When a spectator is forced to perform a task, conspicuous objects do not attract the attention if they are not relevant for the task. In a user study, a high resolution image was compared to both a composite (selective) reduction and a low-quality image of the same sample scene. The high-quality image had a 3072x3072 sampling resolution, and the low-quality sampling resolution was 1024x1024. The selective method generated a composite image where all non-task-relevant objects were taken from the low resolution image, but the important objects were taken from the high resolution. The images were presented to the user, and a questionnaire was completed.

The results show that the participants did not notice any significant difference in quality between high and selective quality images. In a non-task-bound environment, however, all participants found differences between the high and the selective images. Based on these results, a progressive rendering framework for animation is presented. The high level vision model creates a ranking of scene objects depending on the tasks and geometry information available in a so called task-map. The more important an object is rated the more detail will be generated by an image-based rendering method. This is similar to the method presented by Myszkowski [Mys02]. If processing time is left, refinement is performed by accounting preferably for object motion. The authors therefore leverage the method of Yee et al. and enhance it with their task-map.

3.4.4 Drawbacks of 2d Perception-based Approaches

All 2d methods, independent of whether they are based on lower or higher visual processes, have in common that the results need to be mapped on a projection of a 3d object. This projection, however, can alter the appearance of the object and results in hidden or obfuscated features. Furthermore, single features are approximated by the image-based extractors despite the fact that they are directly available through the 3d data. For example, the orientation filters used by Itti et al. [IKN98] extract object boundaries and inner curvatures. This type of information can be extracted directly from the 3d data. This increases not only accuracy, but also efficiency. Curvature can be precalculated and stored for each object, which avoids recalculation of this information in every frame.

The process performed by the HVS during extraction of preattentive features includes spatial information, such as stereoscopic depth, curvature, or form. The methods presented before, however, do not use this information and rely on image-based representations, which imply a mapping of values during rendering. To counteract the rather high costs of calculating the visual salience online, these methods often used global illumination or raytracing solutions. The latter, for example, allows to establish a mapping between an estimated image, which contains the saliency information, and a ray. This way, perceptual information can influence the result, but is very costly due to the raytracing algorithm used. For interactive rendering, a feedback method needs to be devised (see [Dre+07]).

These additional steps can be circumvented when 3d data is directly utilized. As mentioned before, this not only increases the accuracy of the computed perception information, but also the performance of the rendering system.

3.4.5 3d Saliency-based Approaches

For rendering algorithms, it seems obvious to operate on existing 3d features during saliency calculation and not on their projected counterparts. Interestingly, only few researchers focus on this part and the first step towards saliency calculation based on 3d features has been made by Lee et al. [LVJ05]. They proposed a *mesh saliency*, which is calculated as the curvature $\mathscr{C}$ of the objects surface. The extraction is performed using the method of Taubin [Tau95] that extracts the principal curvature on a vertex regarding the neighborhood of the vertex.

Lee et al. accounted for the distance required in saliency calculation by increasing the radius of the neighborhood during curvature estimation. For the calculation of the final saliency value, the authors use the following equation:

$$\mathscr{S}_i(v) = |G(\mathscr{C}(v), \sigma_i) - G(\mathscr{C}(v), 2\sigma_i)| \tag{3.15}$$

where G is the Gaussian-weighted average and σ_i is a predefined scale value depending on the diagonal of the bounding box of the object. The authors use a fixed percentage (0.3%) and derived five consecutive scale values to account for the distance. Here, also the center-surround mechanism is included by calculating the difference between two consecutive curvature values depending on

their scale. This curvature data is provided to a LOD-algorithm to select vertices depending on their saliency value. The lower the value of the vertex, the higher the probability that it will be removed during creation of coarser levels. The presented approach is then applied offline, and thus is independent of the applied scenario.

In figures 3.9b to 3.9e several outcomes of the reduction applied to a object are presented. The images show a comparison between the *mesh saliency* and the QSlim method presented by [GH97]. Some example saliency values are shown in figures 3.10a and 3.10b.

An interesting alternative application for saliency is the selection of viewpoints. Once the saliency of a 3d object has been calculated, the highest saliency value should contain the most interesting features and thus provides the most relevant view of the object. Lee et al. were the first to show this form of application, which was later used by other researchers as well, e.g. Feixas et al. [FSG09]. In the approach of Feixas et al., the saliency of a polygon is defined as the Jensen-Shannon (JS) divergence between neighboring polygons:

$$\mathscr{S}(o_i) = \frac{1}{N_0} \sum_{j=1}^{N_0} \mathrm{JS}(p(V|o_i), p(V|o_j)) \tag{3.16}$$

where o_i is the polygon and o_j is a neighboring polygon. V defines the current view and $p(V|o_i)$ is the mutual information provided from the view and the polygon. This method is inspired by an information theoretic approach where the distributions are interpreted as a mutual information channel between viewpoint and the saliency of the surface.

3.5 Conclusion of Perception-based Rendering Methods

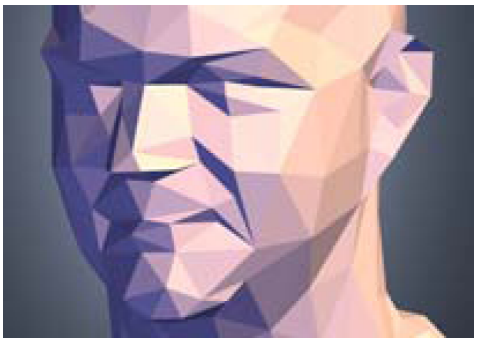
The application of perceptual information in the context of rendering is a wide spread field. It ranges from trivial approximations, plain efficiency aspects (in old LOD-algorithms), to complex computational methods to simulate the HVS. The application of perception-based data introduces a large performance gain in rendering scenarios. It fits well into other available speedup techniques, such as culling methods and acceleration data structures.

However, only small advances in direct extraction of perceptual data in 3d space have been made, and the use of the highly sophisticated models is often restricted to incremental rendering solutions. The latter often implies the use of raytracing or global illumination algorithms. For both, it has been shown that large savings are achieved by utilizing perceptual information. This also holds for early vision-based information, such as saliency. This information enhances the stored LODs, but this data has to be preprocessed. The LODs are selected using a normal distance-based heuristic to avoid expensive calculations of surface features.

Several tests have proven the applicability of perceptual information into rendering algorithms and the viability of a saliency map as an indicator of important regions. The greatest reduction is achieved when using task-driven selective rendering methods. However, it is nearly impossible to assure that the task is performed.

(a) Original version of the head (606K triangles)

(b) 4K triangles with QSlim

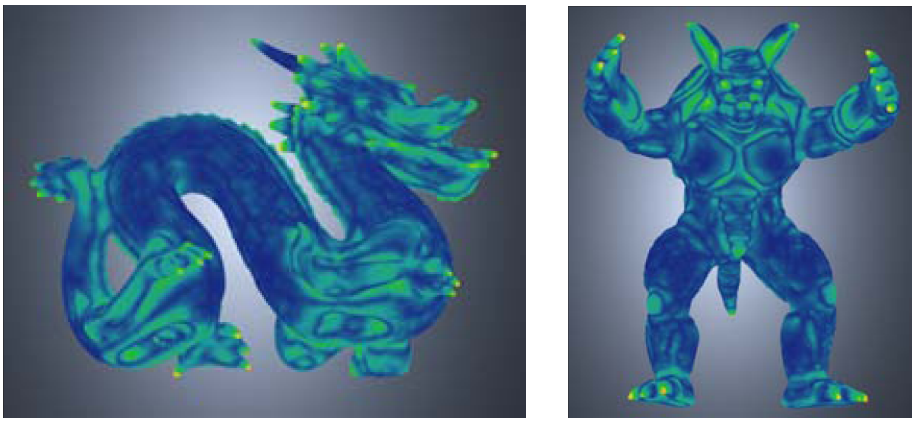


(c) 1K triangles with QSlim

(d) 4K triangles with mesh saliency

(e) 1K triangles with mesh saliency

Figure 3.9: Comparison of the reduction quality achieved by the QSlim [GH97] and the *mesh saliency* approach. The saliency method preserves edges and internal features while the QSlim method removes them.



(a) Stanford dragon (b) Stanford armadillo

Figure 3.10: The extracted saliency values from an object representation. In figure 3.10a, the Stanford dragon, and in figure 3.10b, the Stanford armadillo with their saliency values are shown.

Current methods mainly focus on describing the complete scene. Only the method presented by Lee et al. [LVJ05] allows a object-based adaptation. A universal approach, however, would need a object-based representation to allow combination with non-perception-influenced representations. The ability to change an object should only be available for highly tessellated objects where a gain in processing time can be achieved if a reduction is applied. Evaluating perceptual information for a plane or a simple object may introduce overhead without any chance to refine it.

When evaluating the saliency information, current GPU implementations should be taken into account. By processing the input image, the saliency map can be extracted using only GPU methods, and no data needs to be transferred between the CPU and the graphics card. During this processing, no bottlenecks would be introduced by the rather slow bus-transfer between both.

The combination of 3d data and the GPU saliency calculation would allow fast access to object-specific perceptual information. By processing this data similar to the presented perceptual-based rendering approaches, the representation could be modified based on the extracted perceptual information. The question remains, which processes are needed to create such a perception aware system for non-raytracing or non-progressive algorithms. These topics, however, have not been covered within current publications and additional research is required.

4. Theses

The combination of both rendering and visual perception allows to improve performance while retaining visual quality. Current visual models provide an insight into perception of humans, and some of them have also been applied to rendering. However, they have implicit limitations and cannot be applied to modern rendering methods.

One bottleneck is that most perception models, such as visual salience, are only defined for images, and rendering methods need to create a fast estimate of the scene to evaluate perceptual information. Therefore, raytracing or global illumination algorithms are often enhanced because the extraction of perceptual information is expensive. Recent rendering technologies, e.g. deferred shading, tessellation, etc., increase visual quality and reduce processing time. If saliency information is used as an indicator for early vision, even further savings could be achieved. This increases the demand of a technique that circumvents existing limitations.

This, however, requires a saliency representation based not on images, but on 3d objects to remove the dependency on scene estimations. Also, a data structure is needed that provides a LOD-management and operates on perceptual information. But, the combination of these methods and data structures may not need to create a more visually appealing representation. So, the question arises if a dynamic data structure based on saliency information can create the impression of a highly detailed object when omitting unsalient details.

Based on these observations and requirements, the theses of this work are stated that address each of the issues mentioned. Within this work, solutions will be presented how these problems can be solved or circumvented. We will utilize current methods of visual salience, preattentive feature processing, and computer graphics.

Thesis 1:

The existing visual salience definition based on images can be extended to 3d objects to identify regions of interest on their surface.

Currently, only the work of Lee et al. [LVJ05] made a step towards 3d saliency, but only covers surface curvature as a feature. We assume that the preattentive features for 2d saliency calculation are also applicable for 3d, and they will result in correct identification of regions of interest.

Thesis 2:

Object specific preattentive features are independent of the illumination and can be applied to the saliency calculation in any lighting scenario.

The source of this independence is that surface features are either geometrical or topological

properties. In traditional 2d image calculations, the illumination is part of a pixel, and thus no need for such a separation was necessary. For this reason, the independence has not been proposed before. But, in case of a 3d representation, such a separation of illumination and data increase applicability and allows to reuse them in any illumination scenario.

Thesis 3:

The combination of LOD-mechanisms and the 3d visual salience model allows to retain visual quality of a dynamically adapted object better than plain geometrical approaches.

The last thesis covers the application of the visual salience extraction to 3d objects. After extraction, we need to assure that the regions of interest will retain or even increase the detail of a dynamically adapted object. Here, the adaptation of the object is made online, and the extracted visual salience information is directly applied to the 3d data. A mapping method as proposed by others, such as [Hab+01; Dre+07; YPG01], is not used.

4.1 Definition of Visual Salience for 3d Objects

Instead of extracting saliency values from a estimated scene, a visual salience model for 3d objects provides direct access to perceptual information of the surface. With the validation of thesis 1, an indirection layer can be removed. Most of the current methods do not account for information that is available in 3d space, but estimate this information from a projected scene. Without this estimation and the necessary re-projection, performance and accuracy of a perception-based rendering system is increased.

The work of Lee et al. [LVJ05] made a first step towards a 3d visual salience model, but they only included curvature and omitted other relevant features. We will complete this definition and define an equivalent representation of visual salience in 3d space. This representation includes a scene description because the context provides important information for the deduction of the final saliency values.

The 3d visual salience model enables us not only to calculate the salience of individual surfels of an object, but also allows us to position this object freely in a scene. Unlike existing saliency map approaches, no matching between map and scene objects is required.

The combined saliency of all visible objects in the scene create a representation similar to a saliency map. This allows to apply existing LOD-methods without large changes. Performance is increased by using modern parallel hardware architectures, and parallel processing of the adaptations is performed on each object instead of the entire scene. This is possible because the saliency information is available for each object independently.

4.2 Illumination Independence of 3d Preattentive Features

Preattentive features are obtained from the surface of an object, and thus they are specific to the underlying object. These are influenced by several factors, such as position, distance to the camera, and the light scenario. The features, however, are not removed by these factors, but are rather dampened. For example, colors on the surface of a picture are not changed due to the illumination present in the scene. Only our perception of it may change – a red region does not change its color when light is absent.

With thesis 2, we are allowed to precalculate object related preattentive features and store them in a lookup table for later use. The 3d visual salience model is then applied to this stored set of preattentive features. In combination with an special illumination model, correct or approximately correct saliency values are reconstructed by the 3d visual salience model. To store and access a feature set for an object, a method for looking up these values with respect to the influencing factors is required.

4.3 Perception and Rendering

Perception-based rendering is not a new topic, and saliency has been incorporated also into various rendering scenarios. However, these method either partially account for a complete saliency representation or are limited to a small subset of rendering approaches.

The third thesis joins the before mentioned 3d visual salience model with continuous and dynamic LOD-algorithms. These algorithms are not limited to progressive representations or raytracing methods, and thus they benefit from recent advances in computer graphics regarding the rendering pipeline. By providing visual salience as an indicator for important regions of an object, an algorithm is able to select an appropriate representation, which is optimal in a perceptual manner.

The application of the LOD is not performed during a preprocess. It allows a more fine grained adaption to a scene than any fixed method could. Furthermore, the selection of detail is performed within an object. This inner-object LOD is created dynamically by the rendering framework, and thus the loss of detail is reduced. The displayed representation is optimized for a human spectator.

The adaptation of a 3d model requires a representation that allows dynamic expansion or reduction regarding the object's primitives. Also, a mapping of the saliency values to the current representation is required that allows to select and modify important and unimportant parts.

To assure valid results when applying saliency to 3d data and to confirm theses 1 and 3, evaluations in form of user and performance tests are conducted. The former proves the validity of preserved detail by applying 3d visual salience model in the context of rendering. The latter shows that the introduction of an optimization in the framework does not prohibit interactive frame rates.

4.4 Conclusion

As stated before, an enhanced representation of available LOD data is required, which relates perceptual information to the current representation. We will leverage modern computer graphics methods to increase performance and retain the possibility to change an object in-place. As visual perception is incorporated, the result will provide higher visual quality for a human spectator.

To extract salience of the object's surfels, a link between preattentive features in 2d and 3d space will be established. This allows us to evaluate visual salience for an 3d object while leaving the existing and verified model of visual salience unchanged. Furthermore, we will extended this visual salience model with an illumination, so that precalculated, preattentive features of an object lead to the saliency values of an arbitrarily positioned object in the scene. The position of the light source influences the saliency values accordingly (refer to sections 'Definition of Visual Salience for 3d Objects' and 'Illumination Independence of 3d Preattentive Features').

The combination of both the dynamic LOD-adaptation and the 3d visual salience model defines a full perception-based representation of a single object in a scene. As each object in the scene can be improved using perceptual information, the complete scene representation will be influenced by saliency. The rendering system reduces the amount of primitives while it avoids the introduction of distracting artifacts. It also improves the representation of the objects in the scene with respect to visual measures. Furthermore, it enables us to create dynamic and continuous LOD within an object, and so avoids the generation of LODs in advance.

5. Conception

After having defined current systems and presented our theses, we now take a first step towards the incorporation of perception in 3d rendering. More precisely, a framework will be defined that accounts for evaluation of saliency information and its application to an individual object. This includes the adaptation of existing visual salience calculation methods, presented in 'Perception-based Rendering' on page 51, to a 3d scenario. Furthermore, an abstract, dynamic LOD-representation will be presented to process the provided visual salience information.

Our dynamic system framework is based on a Model View Controller (MVC)-pattern commonly found in rendering applications [SMA10]. This allows the presented methods to be integrated in existing applications without the need of major modifications. The *Model* and the *Controller* are extended with the saliency information and the respective processing capabilities. The *View*, however, remains unchanged.

5.1 The Perception-based Rendering System

The common denominator of the presented methods is the rendering system. Representations can be selectively enhanced using visual perception, thus allowing individual scaling. The system consists of two main stages: a *Preprocess-Stage* and an interactive *Feedback Stage*. A complete view of our proposed rendering system is given in figure 5.1.

The first stage, the *Preprocess-Stage*, extracts preattentive features and stores them for later use. A LOD-representation is generated, resulting in a multi-resolution object. If both multi-resolution and features are stored in a file, the *Preprocess-Stage* is reduced to a simple loading of the available data.

The other part is the *Feedback Stage*, which extends the MVC-pattern. As the current information

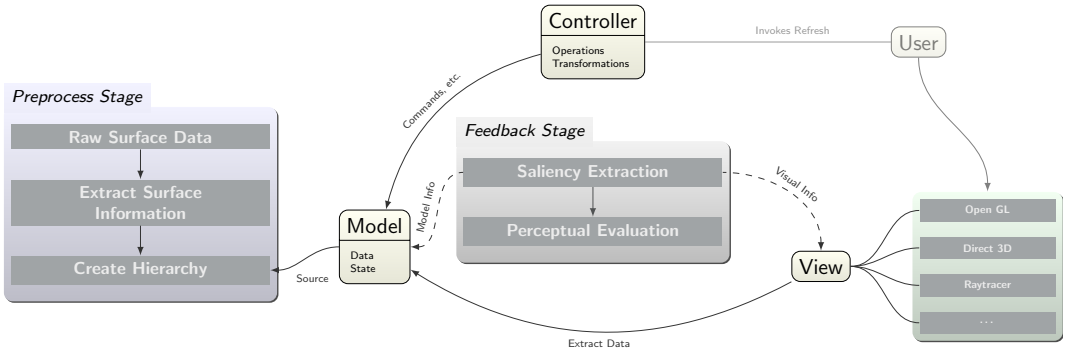


Figure 5.1: The overall view of our rendering system. A basic MVC-design is extended by adding a *Feedback-* and *Preprocess-Stage*. The dashed lines show optional processing paths, e.g. it is not necessary for the *Feedback Stage* to issue a change. The arrows denote the the process flow.

regarding the current object may be changed by the *Controller*, which processes the input from the graphical user interface, the representation may need to be updated. In case of an update, a new *View* is created, and new visual salience information is available. This may lead to another change to the system without any user interaction. So, a *Feedback System* emerges between saliency processing, given by *Controller* and *Model*, and *View*, which produces the visual output.

To effectively control the behavior of this dynamic system, a dampening mechanism has to be introduced to stabilize the object representation. This dampening mechanism needs to be defined by the application and should be located in the *Controller* or the *Feedback Stage* because these invoke the changes made to the *Model*.

5.2 Preprocessing 3d Objects

It is common in real-time rendering applications to defer computations or reuse results whenever possible. The time needed for the generation of frames needs to be minimized. Therefore, the *Preprocess-Stage* calculates all information that can be processed without scene information. This includes information required for a LOD-algorithm and the preattentive features used by the *Feedback Stage*.

5.2.1 Level of Detail Generation

As an object can be placed anywhere in the scene, a predefined set of LODs may not be sufficient. When creating perception-based rendering system, the representation needs to be adapted with visual information. Thus, the 3d object needs to allow a change in LOD in dependency on the required detail. These details are not fixed to a certain set of LODs.

Basic multi-resolution representations provide solutions and algorithms to select or manipulate the LOD accordingly. To avoid on-the-fly computation of the intermediate levels, the LOD-data is generated during the *Preprocess-Stage* and can be accessed in the interactive *Feedback Stage*. The LODs are generated using one of the existing approaches presented in section ‘Level of Detail’ on page 26.

The LOD data or, more precisely, individual primitives need to be accessed often by the *Feedback Stage*. An acceleration structure, such as a tree, would allow to select the necessary data fast and can provide relational information, e.g. successors or predecessors. The selection is required to dynamically control the appearance of the object with respect to both the LOD data and the perceptual information.

5.2.2 Preattentive Features

As introduced in ‘Visual Perception and the Human Visual System’ on page 11, the main idea of visual salience modeling is to compute preattentive features, which are based on early vision in the HVS. The architecture presented by Itti et al. [IKN98] provides a method for the generation of saliency maps based on 2d input images.

The calculation of saliency information, i.e. applying image filters by convolution with the source image in order to extract information, depends heavily on neighbor information in image space. In real-time applications, these calculations may not be affordable as they consume precious calculation time. Instead of calculating saliency information on the fly, most of the data is precalculated, based on thesis no. 2.

A nearest neighbor search among surfels is required to extract these preattentive features. This step can be as trivial as extracting edges in a mesh-based representation up to a complex operation if a point-based representation is used. If topological information is not available, a nearest neighbor algorithm, such as the z-order curve, can be applied to increase overall performance (refer to 'Calculation of Geometric Properties' on page 33). Once topological information is available, geometric and color-based information can be extracted without applying an illumination. Referring to thesis no. 2, the illumination has to be accounted for during the saliency calculation and not during extraction of the preattentive features.

The results of the feature extraction should be stored in a GPU-friendly data layout, such as textures. In combination with a shader-program, this data is then be available during rendering and a combination of the preattentive features can be performed with object position and illumination information to derive the final saliency values.

Requirements for the *Preprocess-Stage*

- LOD-data is generated in form of multi-resolution objects
- Acceleration structure can allow fast access to individual primitives
- Surfel neighbor information is required for preattentive features
- Preattentive features are stored independent of illumination
- GPU-friendly layout for preattentive features allows efficient access during rendering

5.3 Feedback Stage and Perceptual Evaluation

The *Feedback Stage* incorporates a calculation of visual salience along with interactive modifications of an object. The *Dynamic Data Structure* utilizes the 3d data created during the *Preprocess-Stage* and changes the LOD accordingly. An identification of primitives to change is performed by the Perceptual Evaluation where saliency values are compared.

The *Controller* receives an input from the user, which is then forwarded to the *Model*. Based on this input, the *Model* creates a updated representation and a new *View* is generated. Once the perceptual evaluation is enabled, the feedback mechanism becomes active and the representation of the object is changed according to the visual salience information. The *Feedback Stage* therefore issues the appropriate commands to the *Controller* (See figure 5.1).

5.3.1 *Dynamic Data Structure*

The *Dynamic Data Structure* is based on the 3d multi-resolution data structures generated during the *Preprocess-Stage*. This structure allows to change single primitives within the representation. This change is invoked externally by the *Controller*. The *Dynamic Data Structure* is part of the *Model*, which is used to derive the *View* and the perceptual information used by the *Feedback Stage*.

The *Dynamic Data Structure* is not limited to a specific type of primitive, and thus the rendering method can be selected accordingly. A support to change the LOD of a given primitive is required using solely the multi-resolution information. In case of meshes, a more detailed mesh must be generated while a voxel has to be subdivided into multiple voxels that describes the complete volume of the parent. Within this thesis, a point-based rendering method is used, which removes the necessity of recreating volumetric or surface-related information. As the source data may become extremely large, the data structure may need to operate in an out-of-core manner.

The change of the LOD is not limited to geometric properties, and even an exchange of the illumination model is possible. For example, a transition between two geometric LODs may also include a change of the illumination algorithm. This selection is handled internally, thus reducing the influence of external components to a minimum.

Required Capabilities of the *Dynamic Data Structure*

- Operates on multi-resolution and acceleration data
- Increase/Decrease detail of a single primitive
- Supports out-of-core capabilities
- Multiple types of detail are supported, e.g. illumination and geometric detail

5.3.2 Perceptual Evaluation

The rendering process creates the current view of the scene based on light sources and camera position. Furthermore, all necessary saliency information of the objects in the current representation is provided. The optimization of the current view begins with the selection of primitives using saliency information. This is acquired from the object and the current view. A command for change is issued and is forwarded to the *Controller*. To avoid local maxima expansion, multiple primitives are processed before an update of the saliency information is requested.

The evaluation includes the extraction of preattentive surface features created either during the *Preprocess-Stage* or by an on-the-fly computation. The feature values are combined and weighted during rendering of the object. This forms the final saliency values. If the evaluation is not performed on the graphics card, this data must be transferred from the GPU to the CPU.

The selection of the primitives is an optimization problem and is restricted either by user-defined or system-related limitations. These limitations are, for example, a maximal number of primitives

to be drawn and do need not to be static during run-time. It is also possible to derive limitations by defining an upper bound regarding the processing time, processed primitive count, or frame rate. Therefore, the *Feedback Stage* can adapt the rendering to various types of limitations.

The optimization algorithm alters the representation of the object to the highest detail possible. This detail is defined with respect to perceptual information and the imposed limiting factors. If no restrictions are given, the algorithm will converge towards the highest LOD in the multi-resolution and no further evaluation is required. Once a restriction is imposed, the solution will become an approximation. In real-time applications, a perfect result may not be achievable. Thus, similar to other online algorithms, a greedy evaluation method is most suitable for the presented system because a fast selection of primitives for display is required. To allow further speedup, the evaluation algorithm may be distributed over multiple frames. So, it is possible to defer changes of the object's representation or refine the selection of primitives for modification.

Operations Performed by the Perceptual Evaluation

- Combines information from rendering and object
- Optimizes the representation using individual primitives
- Various types of limitations can be imposed
- Selection of primitives is performed online

5.3.3 Dynamic Rendering System

The *Feedback Stage* and the *Dynamic Data Structure*, provided by the *Model*, form a dynamic rendering system that accounts for visual perception. As this system should maintain interactive frame rates, expensive operations must be avoided. Therefore, all possible calculations have to be performed before the interactive rendering has begun. To avoid stalling of the drawing mechanism, rendering and perceptual evaluation are distributed over multiple threads. This, however, is not an integral part of the system and can be bypassed if multi-threading is not supported.

When applying an optimization algorithm, a repeated change between two LODs can occur. This repetition, however, will result in a highly salient flicker. As the transition is not a part of the perception model, it is also not recognized by the evaluation. Thus, the dynamic system needs to incorporate strategies to quickly resolve such issues. Additionally, smooth transitions between different LODs can reduce the effect of the flicker.

The presented system is optimized for a single object, but can easily extended for multiple objects as well. The Perceptual Evaluation in this framework is not bound to a single *Dynamic Data Structure*. When attaching multiple objects, the optimization method will select the most important primitives and a change in detail is executed in the assigned *Dynamic Data Structure*. In this case, the algorithm behaves like scene-based perceptual rendering methods.

Requirements of the Dynamic Rendering System

- Combines *Feedback Stage* and *Dynamic Data Structure* to account for visual perception
- Parallel computation of drawing and evaluation
- Optimization-related issues need to be resolved quickly

5.4 Next Steps

Based on the conceptual framework presented in this chapter, the creation and implementation of the *Dynamic Data Structure* is discussed in chapter 'TreeCut' on page 73. It will serve as a starting point for the incorporation of saliency features in a 3d rendering system. Some Perceptual Evaluation methods are defined as well, which perform the selection of primitives of the current object's representation.

Afterwards, a saliency function will be introduced in chapter 'Bidirectional Saliency Weight Distribution Function' on page 99. This will be combined with the *Dynamic Data Structure*. This function is either evaluated during the rendering process or its results are stored in memory based on precalculated scene object views. In contrast to other methods, individual object features are taken into account instead of features derived from images of the entire scene.

The dynamic rendering system will be implemented to adaptively change the LOD representation of a 3d object. Therefore, multiple extensions have to be included to allow an interactive processing – Details are given in chapter 'Feedback System' on page 125.

6. *TreeCut*

Due to the independent nature of point-based representations, a multi-resolution object is acquired easily. A selection of a specific level within this multi-resolution representation is usually based on distance to the camera, point-size, or a combination both criteria. These criteria are inspired by human limitations and are intended to reduce overdraw if points start to overlap. For example, the reduction of geometric detail in the periphery or far distance reduces the overall number of primitives drawn while being almost unnoticeable by a spectator.

The selection of nodes is made either during a traversal of a tree or during application of operations on each surfel in case of a progressive representation. To increase separation from the underlying data structure and to allow a more fine grained selection of primitives, an abstract data structure is necessary. The presented *Dynamic Data Structure* allows to select nodes using only the current representation, and it also avoids unnecessary re-traversal of the multi-resolution object each time it is drawn. Inspired by graph cut approaches, the *TreeCut* operates only on a set of active nodes from which a closed surface of the object is derived.

In the following sections, we will define the *TreeCut* and compare it to other, similar techniques. The individual operations, which will be performed on the current cut, are presented and discussed in section 'Dynamic Strategies'. The implementation provides a GPU-friendly layout, which is presented in section 'Implementation'. Based on this prototypical implementation, results are presented that show the universal application of the presented *Dynamic Data Structure*.The chapter concludes with a summary of the proposed methods and achieved results. Additionally, we show where the current prototype and definitions can be enhanced even further.

6.1 Definition of the *TreeCut*

In the following definition, the *TreeCut* is applied to a tree hierarchy, which contains different LODs of an object. It is also possible to maintain a *TreeCut* that is based on a progressive representation.

The active set of nodes, the cut-nodes, is generated by a traversal of the provided tree. In a path from the root towards a leaf, only a single cut-node is present and the collection of all these cut-nodes defines the *TreeCut*. This assures that a node of the tree is not covered by multiple cut-nodes. The operations presented in section 'Methods Used in the *TreeCut* ' will always result in a valid *TreeCut*. A mathematical definition of the *TreeCut* is as follows [SK10]:

Definition 1 (*TreeCut*)**:**

Let P be all paths from the root to the leaves of a tree. When there is exact one cut-node C on each individual path

$$\forall p \in P : |\{\text{node} \in p | \text{node is cut}\}| = 1,$$

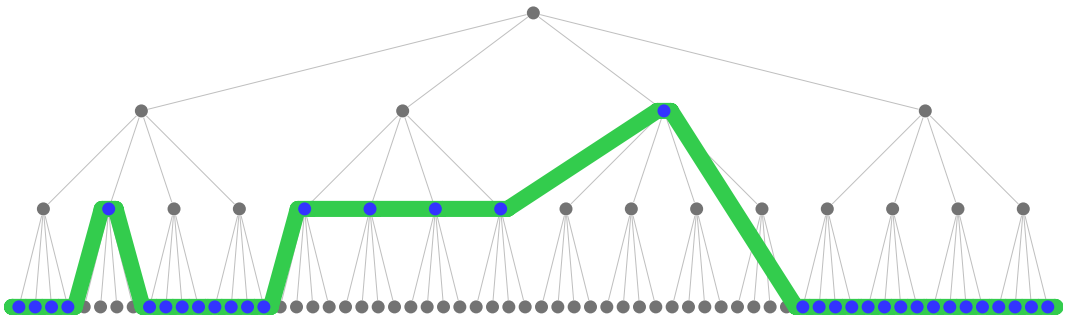


Figure 6.1: A *TreeCut* based on a tree representation. The green background highlights the cut-nodes. Note that the cut spans multiple levels, and the surface is extracted solely from the individual cut-nodes.

then the *TreeCut* is defined as the set of all cut-nodes C:

$$TreeCut = \bigcup_{C \in \text{tree}} C$$

The *TreeCut*, or in short cut, is modified by two core operations, namely `refine` and `coarse`. Their names are due to the change they cause in the rendered object: a `refine`-operation increases detail while a `coarse` reduces it. The *TreeCut* can be initialized with a recursive algorithm starting at the root node of the representation and is aborted at a predefined level. After this recursive traversal, no further traversals are required, and the operations on the cut only require node information, such as pointers to parent and child nodes. For rendering, a node needs a representative surfel. An example *TreeCut* is depicted in figure 6.1, showing both a tree and a *TreeCut* generated upon it.

We propose to use a double-linked list for storage of the individual cut-nodes because no fixed connection between individual cut-nodes exist. The list representation allows efficient removal and insertion ($O(1)$) if a node has been selected. This selection, however, requires linear time ($O(n)$) because only the head and tail of a list are directly accessible. This is not a drawback when the complete *TreeCut* is evaluated at once. Additionally, the data of a cut-node does not need to be copied, i.e. the *TreeCut* uses the underlying data structure as a data provider.

For rendering, a closed surface representation may be necessary. Therefore, we give another definition regarding properties of the underlying data structure.

Definition 2 (Cut-Criterion)**:**

All nodes need to cover the complete area or volume of their successors.

For example, in a tree, the Cut-Criterion is fulfilled if the parent node covers its children. Therefore, usually the size of the parent is defined as the maximal displacement from the parent to the children plus their respective size. If this is assured, each *TreeCut* will generate a closed surface representation.

6.1.1 Difference to Other *TreeCut* Methods

The idea of a cut through a tree to represent points or a surface is not new, and each of the cuts is a LOD-version of a multi-resolution object. Our presented *TreeCut*, however, varies from other definitions by providing a sharp representation of the cut and enabling operations explicitly for modification.

A traversal algorithm, such as the QSplat-based rendering, generates a cut not explicitly, but it emerges by aborting the recursion. At these nodes, a cut is defined. However, no direct control is possible.

Sequential Point Trees, as an extension to the QSplat rendering, are a linearized version of the recursive algorithm [DVS03]. But again, no cut is explicitly defined because an abort criterion is used. Thus, no operations, which would alter the current cut, can be applied. Furthermore, the cut itself is not sharp, meaning a cut-node can span multiple levels at once as a vertex shader may choose to discard a node during rendering.

Kalaiah and Varshney [KV03] employed a tree cut to allow a client-server transmission of point-based data. On the one hand, the data structure provides basic split- and merge-operations to allow dynamic manipulation. On the other hand, no strategies are defined to select cut-nodes and apply these operations to them. This, however, is crucial for our perception-influenced rendering.

Recently, Carmona and Froehlich [CF11] proposed the “optimal octree cut for high efficient voxel rendering”. Their data structure was presented simultaneously to our approach, and the main idea of retaining the cut and directly operate on the representation was stated independently in both publications. Carmona and Froehlich additionally present a theoretical optimal strategy for managing an octree cut. In our work, we extend their ideas, such as the usage of a maximization problem for optimizing the *TreeCut*-representation.

6.1.2 Role within the Framework

In our framework, defined in chapter ‘Conception’ on page 67, the *TreeCut* is an integral part of the *Model*. The representation of the underlying multi-resolution data structure is extracted only from the current cut similar to Carmona and Froehlich [CF11]. In addition to the plain rendering, the *Feedback Stage* modifies the *TreeCut* via the *Controller*. Therefore, a strategy is required that allows to modify the current representation. The *TreeCut* then processes the input made available by the *Controller*, which relays the changes regarding the scene to the *Model*.

The strategies, which apply changes to the object’s representation, are part of the Perceptual Evaluation and are covered in section ’Dynamic Strategies’. This evaluation is not particularly bound to a single object, but the current approach only covers this case. Multiple objects are accounted for, for example, by incorporating a job-scheduling algorithm. The individual evaluations would then be invoked in dependency of their internal priority.

In figure 6.2, the enhanced *Model* of our framework is depicted. As the *TreeCut* is not restricted to be unique, several instances can be created based on the data provided by the *Model* to enable

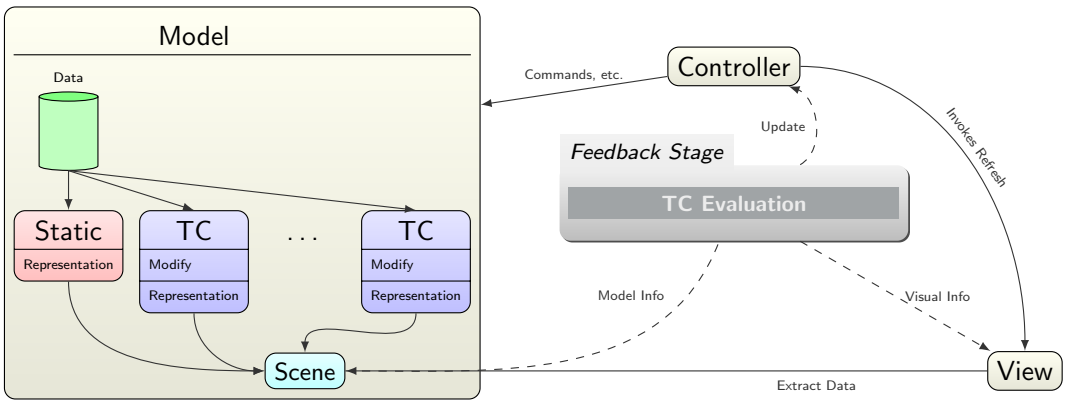


Figure 6.2: A detailed view of the *TreeCut* within our framework. It extends the *Model* and operates on 3d data with the ability to modify the representation. The *Controller* invokes the necessary operations while the *Feedback Stage* extracts the nodes to be affected by an operation. In combination with static objects, the scene is generated and used for the *View* to derive a visual representation.

dynamic LOD on these objects as well.

6.2 Methods Used in the *TreeCut*

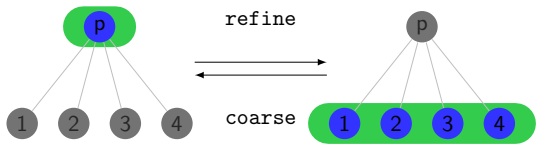


Figure 6.3: The main operations defined by the *TreeCut*. The operations are independent of the representation and only operate on the current cut.

An initial cut can be generated, for example, with a recursive traversal algorithm, which ideally fits to any tree structure. Either a level-based, a priority-based abort criterion, or a previous cut is used to generate a new cut. We will apply the data structure mainly for priority optimization, but other applications and evaluation mechanisms are also practicable, e.g. to create a stippling pattern. Stippling is a “traditional pen-and-ink technique that is popular for creating illustrations in many domains" [Mar+10]. Some results regarding this technique are presented in section ’Stippling Results’.

Once the cut is generated, we propose two different update strategies to modify the *TreeCut*. The first calculates a priority for each cut-node and maximizes or minimizes the overall priority of the cut by applying the *TreeCut*-operations. The second determines a target-level for each cut-node and performs an appropriate *TreeCut*-operation.

6.2.1 Altering the *TreeCut*

To modify a *TreeCut*, two core operations are applied: The `refine`-operation expands a selected node and replace it with its child nodes. The inverse, the `coarse`-operation, replaces all child nodes with their common ancestor, i.e. the parent node. In figure 6.3, both operations are visualized.

The `refine`-Operation

Assuming a valid *TreeCut*, any cut-node can be selected for the `refine`-operation. If the cut-node is a leaf node, then there are no successors regarding the hierarchy, and the operation aborts. Otherwise, the `refine`-operation removes a cut-node and inserts its child nodes. By appending the children next to the parent node before deletion, all children are stored in sequential order. The child nodes are marked as cut-nodes, and the parent is removed from the double-linked list.

The new representation, again, is a valid cut because no additional nodes have been skipped or removed. As the parent has been deleted from the cut, the cut is re-established by the insertion of the child nodes. Also, a closed surface is generated, assuming that the Cut-Criterion is fulfilled.

The `coarse`-Operation

The `coarse`-operation removes all child nodes and replaces them with their common parent node. This parent is then added to the cut, which recreates and re-validates the cut (see definition (1)). As the parent is inserted before the first child in the list and the siblings have been added in sequential order (refer to the `refine`-operation), the removal of the last child connects the remaining cut with the parent node.

Unlike the `refine`-operation, the `coarse`-operation may not be applied to every cut-node without fulfilling the following condition: All siblings of the current node are required to be cut-nodes. Otherwise, the `coarse`-operation removes more nodes then intended because sub-trees could be removed. In addition, one has to make huge efforts to clean up and remove invalid nodes within the cut representation. To avoid both, the `coarse`-operation is limited to so called `coarse-parent` nodes, which are defined as follows:

Definition 3:

A node is a `coarse-parent` iff all its direct children are part of the cut.

If this precondition is met, the `coarse`-operation can be applied to the *TreeCut* without any harm. To allow fast extraction of `coarse-parents`, we propose to keep track of the number of cut-nodes in the parent node. Then, the decision whether a node is a `coarse-parent` is then reduced to a comparison of the number of children of the parent against the number of children in the current cut.

6.2.2 Dynamic Strategies

Based on the presented two core operations, different strategies to evaluate and change a cut are possible. As stated before, we focus within this work at an optimization and a *target level* approach. The former updates the cut to represent an approximative optimal solution using some restrictions, such as size, whereas the latter changes the cut to meet a certain distribution.

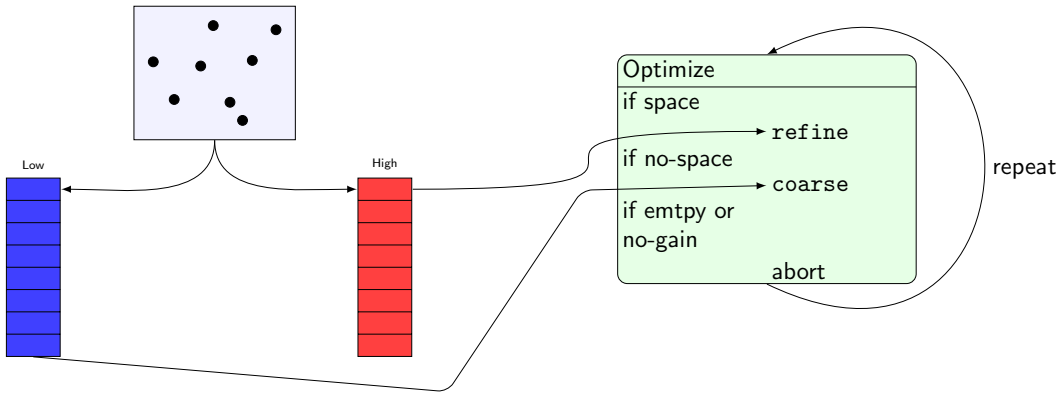


Figure 6.4: Optimization performed during evaluation of the *TreeCut*. The current surfels are extracted from the *TreeCut* and sorted in low- and high-buffers based on their priority. The optimization `refine`s or `coarse`ns depending on the size of the representation.

Cut Optimization

For the optimization approach, the cut itself is given a weight, and each operation changes this by either increasing or decreasing it. In the case of priorities, the cut will be optimized by `refine`-ing important and `coarse`ning unimportant nodes. These operations are subjected to a restriction, such as to retain a given size. This way, the overall weight of the cut is increased as a combined application of `refine`- and `coarse`-operations expands important nodes and thus increases the weight. In this case, we assume that the weight of the parent are greater than or equal to the summed weights of the children.

Figure 6.4 shows a flow chart of the evaluation process. First of all, the cut is created by filling in the nodes using the nodes from the high buffer until a maximum size is reached. Based on this state, the cut is sorted by the priority values of each node in ascending and descending order. The most unimportant nodes are `coarse`ned to free up space within the cut, and this new space is filled with the `refine`d result of the most important cut-nodes. These steps are repeated until an abort criteria is met. Suitable are conditions, such as that no nodes are `refine`-able anymore or any pair of operations will not increase the weight of the cut. The latter is the case when a `coarse`-operation reduces the weight more than a `refine`-operation will increase it. Additionally, other criteria can be added, for example, a time restriction, which is well suited for fast adaptation in an interactive scenario.

To avoid an expansion of the most important node only, several nodes are processed before updating the internal sort result. This results in a more homogeneous application of the `refine`- and `coarse`-operations. This approach is inspired by Haber et al. [Hab+01] who stated that a *winner-takes-all* approach is not appropriate. When the number of operations performed on the cut is restricted, a partial sorting improves performance. Instead of a $O(n \log n)$ theoretical time complexity

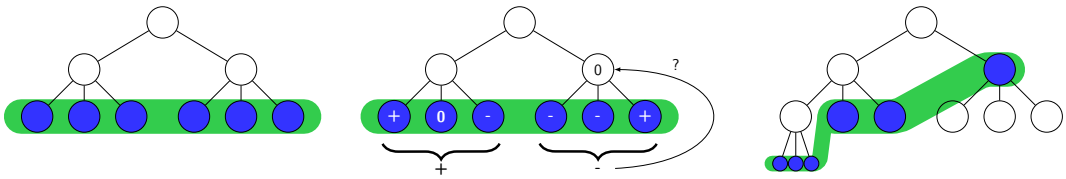


Figure 6.5: The bucket evaluation strategy for the *TreeCut*. Each individual node is examined, and a *target level* is extracted. Depending on the siblings, the operation is selected. If a `coarse`-operation is required, the parent node is also inspected.

for sorting the cut with size n, it is reduced to $O(n \log k)$ where $k << n$ (refer to Carmona and Froehlich [CF11]).

Also, instead of sorting cut-nodes with decreasing priority, which would be used for the `coarse`-operation, only `coarse-parents` are sorted. As the number of `coarse-parents` is smaller, an additional speedup is achieved. This approach is also proposed by Carmona and Froehlich [CF11]. In this case, the `coarse`-operation is applied directly by replacing the child nodes with their respective `coarse-parent`. No extraction of the parent node has to be performed, and only the child nodes have to be selected in the cut representation. If the child nodes are inserted sequentially and a parent node is a `coarse-parent`, then we only need to extract the first child node. All following nodes in the cut necessarily will be siblings regardless of any `refine`- or `coarse`-operations applied before as long as we have a `coarse-parent`. When the underlying data structure is a tree, the extraction can be performed efficiently as child pointers are available. A check for the `coarse-parent` property, however, needs still to be performed during evaluation. Due to previous `refine`-operations applied, the selected node may not be a valid `coarse-parent` anymore.

Bucket-based Distribution

If a priority-based distribution is not applicable, a level-based approach can be chosen instead. Here, each node is given a certain *target level*, i.e. it is assigned to a special bucket. This bucket determines, which operation has to be applied to this node. In figure 6.5, this assignment of the buckets and the result after completion of the strategy is shown.

The procedure traverses the complete cut and computes a bucket for each individual cut-node. Based on the difference between the assigned bucket and the current level, the appropriate cut operation is executed.

The `refine`-operation is only applied if the average level delta of all siblings is greater than or equal to zero. Otherwise, a `coarse`-operation will be applied. In addition to the `coarse-parent` criterion, the `coarse`-operation is subject to the following condition: The parent node's bucket must be less than or equal to the current one. This avoids flicker due to an immediate `refine`- followed by a `coarse`-operation or vice versa. In the bucket strategy, the `coarse`-operation has a higher precedence than the `refine`-operation. Furthermore, no explicit `coarse-parents` are stored and must be checked prior to each `coarse`-operation.

The *TreeCut* will approach a distribution similar to the level values provided by the bucket function. In the case that the `coarse`-operation can be applied in all situations, the *TreeCut* can be equal to this distribution.

It is also possible to iterate the cut multiple times to reach the imposed distribution faster. Also, a termination criteria can be used, e.g. a maximal number of processed nodes, to assure an upper bound of operations to be performed.

6.2.3 Data Separation

The *TreeCut* data structure is intended to be efficient on modern GPUs. This is achieved by storing the geometry data directly within the graphics card memory, so that a necessary fetch of the data is performed as fast as possible. We therefore separate the object geometry from the tree information. The proposed distribution of the data is visualized in figure 6.6.

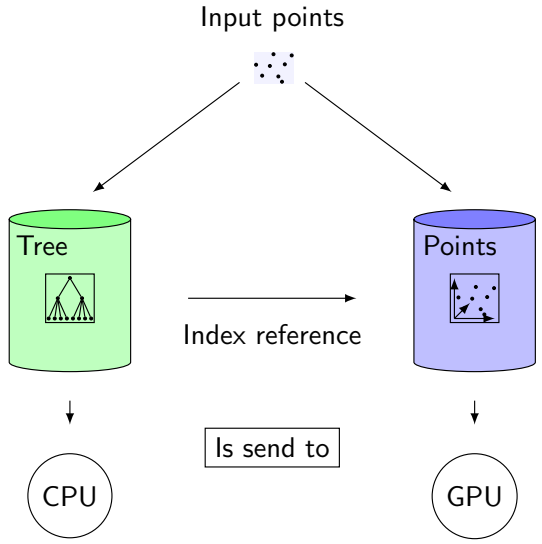


Figure 6.6: The separated data layout utilized by the *TreeCut*. This separation allows to parallel process the current representation while the old representation is drawn by the rendering algorithm.

The geometry data is stored in a Vertex Buffer Object (VBO), which directly maps to the graphics memory. This allows efficient rendering as only the internal memory is used, and large speedups are achieved. This has been pointed out in the "Sequential Point Tree" representation, and it reduces the load of the CPU during rendering [DVS03].

The *TreeCut* also supports an out-of-core memory transfer, but as this is widely researched topic, no special implementation is described in this work. The reader may be referred to the work of Carmona and Froehlich [CF11] where an out-of-core support for an octree cut is presented.

All necessary information for application of the core operations is stored within the nodes and is made available in the main memory. The nodes are referenced via an index and thus allow a straightforward serialization. This is because they are independent from memory format and references to external information.

The distributed storage enables parallel manipulation of the cut. For rendering, an index-list is used to identify primitives to be drawn. An evaluation strategy may also alter the cut without any interference to the rendering. The update of the index-list, however, must be taken care of. This is done by managing two lists. One is modified by the evaluation strategy while the other is used in the rendering process. When the evaluation has finished processing, its index-list is copied, and the rendering operates on the altered list.

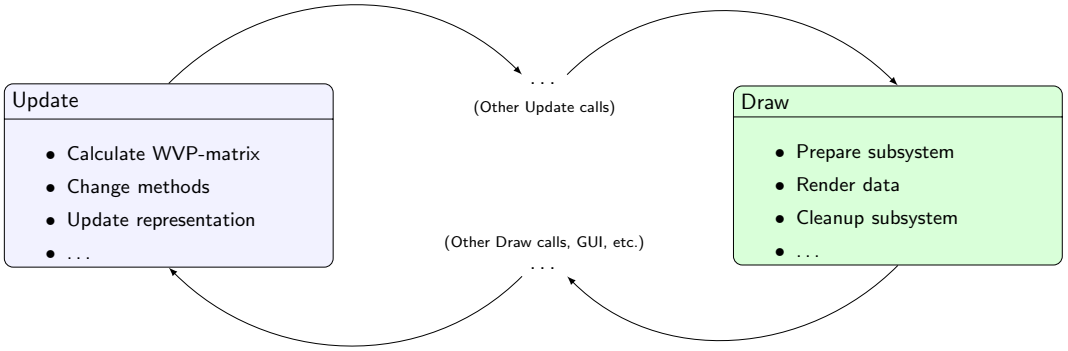


Figure 6.7: The render loop for the *TreeCut* with the internal operations performed during each frame. In the `Update`-method, the local data of the object is recalculated and possible changes are applied. In the `Draw`-method, all necessary calls to the rendering subsystem are issued and the object's representation is displayed.

6.3 Implementation

The main implementation is based on an `Update - Draw` loop, commonly found in rendering applications and game engines, such as XNA[1]. In figure 6.7, a picture of the used `Update - Draw` loop is shown including the internal processing during this loop.

Such a loop is well suited for creating interactive applications with either a fixed or an unbound frame rate. Additionally, object-oriented-programming is directly supported because only the two main methods must be provided by each inherited class. The frame rate is given by the number of `Draw`-calls per second. In the case of a fixed frame rate, either a pause function can be utilized or the `Update`-method is called multiple times to stall the system a certain time until a new `Draw` needs to be dispatched. In the presented framework, we selected an unbound approach.

During the `Update`-method, the cut's representation is recalculated regarding position or orientation. This includes a change of the drawing method, a translation, or a rotation dispatched by the *Controller*. Also, the data between the evaluators and the *TreeCut* is exchanged safely as during the `Update` no drawing occurs. After an `Update`-step, the *TreeCut* represents the most current version, which is ready for display via the `Draw`-method. The actions to be performed are encoded as states, which allows to toggle them independently, and allows external methods to safely alter the representation.

In the `Draw`-method, the current cut is rendered based on an index-list. This list is created during the generation of the cut and is maintained by both core operations (`refine` and `coarse`). The data for rendering must be available at this moment, so that all of the object's surfels are displayed. During a `Draw`-call, no other process may write to the rendering data.

[1] Microsoft XNA Framework

6.3.1 Data Structure

The *TreeCut* is based upon two main data structures, namely a surfel-buffer and the multi-resolution data structure, in the following node structure. This node structure stores the connection and LOD-information. Additionally, each node has a representative in the surfel-buffer.

The surfel-buffer, which resides inside the graphics card memory as discussed in section 'Data Separation', contains only information needed for rendering, e.g. position, surface normal, and other attributes, such as point-size or color. To allow fast access from the node structure, an index-based lookup is established. The buffer is uploaded to the graphics card and may be discarded after successful transmission. To access the data, the graphics memory is mapped using methods of the rendering subsystem. However, the access to this data should be avoided or, at least, batched together because the transfer has a large impact on the performance. The final rendering is performed with an index-list that references the individual surfels in the buffer.

The node structure is based either on a tree, which is made available during creation of the *TreeCut*, or a progressive refinement structure. The only requirement is that a mapping between the surfels and the node structure can be established. The node structure then stores the necessary information for application of the `refine`- and `coarse`-operations. In the implemented prototype, a tree with branching factor 4 was chosen based on the QSplat format [RL00].

The creation of such a tree starts with a set of leaf points, which are partitioned in space with a kd-tree approach. If the set of points is less than or equal to the branching factor, the current node is marked as a leaf node and the recursion is aborted. We do not create a node for each leaf point, but instead, we directly attach the points to a leaf node following the approach of [RL00]. Each node generates a new surfel within the surfel-buffer that approximates the properties of all child surfels. This assures the Cut-Criterion. As a reference to the surfel, its index is stored in the node.

During generation of the *TreeCut*, this tree is traversed with an abort criteria and cut-nodes are stored in a linked list, which allows linear traversal. The cut is sufficient to extract the index-list because each node stores the index to its representative surfel.

The double-linked list allows to insert and remove cut-nodes without the need to manipulate the data structure on which the *TreeCut* is based. To alter a cut-node, only information of this particular node is required, and the result of an operation directly replaces the affected node.

Sorting the Priorities

As mentioned in section 'Dynamic Strategies', a partial sort may be sufficient for sorting the nodes by their priorities. To avoid operations to be performed directly on the double-linked list created by the *TreeCut*, an additional sort buffer, with a fixed size of k, is defined, which allows efficient sorting of nodes based on a priority value.

Due to the sort buffer, the *TreeCut*'s linked list can be optimized for the cut operations. When it is assured that the successors are siblings, i.e. they are on the same level, the extraction is reduced to a simple traversal. If this not the case, the parent node has to be extracted from the node-structure

followed by an extraction of its children.

During evaluation of the cut, the nodes are then extracted from this sort buffer. The maximal number of processable nodes, however, is limited to k. If more nodes are required during an evaluation iteration, a change of the parameter k and an according resize of the buffer needs to be performed.

The default `stl C++`-classes, however, yield a large performance penalty. In the following, we will discuss this issue. Most of the data is copied and freed very often. The priority queue performance was increased dramatically by replacing it with a manual written implementation. The default `std::vector`-container resulted in an update time for the optimization algorithm of approximate 250ms using a partial sort size of 64 elements. The manual implementation performs the same operation in approximate 2ms using the same parameters. For the measurement, a total of 35286 nodes were given as input to the sorting routine.

The reason for this large performance impact is due to the dynamic resize performed within the `std::vector`-class. The container is initialized to a default size of 10 in Microsoft Visual `C++`. As the vector includes an automatic memory management, the internal buffer is resized once the maximum capacity is reached, and the new size for the container is calculated by:

$$\text{size}_{\text{new}} = \text{size} + \frac{\text{size}}{2}$$

A new data block is allocated with the calculated size, the old content is copied to the new data block, and finally the old one is deleted. To allocate enough memory to hold the target size of 64 elements, this *allocate - copy - deallocate* has to be performed 5 times.

When the internal buffer is created with an initial size, which is an option for the `std::vector`, the time needed is reduced to approximately 6ms. This time, only one allocation of the memory is required during each evaluation iteration because a call to the `clear`-method always deletes the elements of the container. The manual implementation improves this by invalidating the content without any deletion when calling the `clear`-method. Therefore, it is important to assure that no memory is allocated by the constructor of the base object. This is the case for the data used (a floating point priority and a pointer to the corresponding node), and thus the performance increases even further, yielding the 2ms stated before.

The evaluation is performed in parallel during rendering because the processing of nodes and surfels is independent of the rendering. Only the index-list, which is invalidated during evaluation, must be handled with care and has to be copied with an exclusive lock. A more detailed view on multi-threading with the *TreeCut* is presented in section 'Multi-Threading'.

Index-list Manipulation

The index-list is updated during the individual calls to the `refine`- and `coarse`-operations. Instead of recreating a new list and traversing the *TreeCut* after the operations are performed, the surfel indices are maintained externally, so that an index value is available without traversing the index-list again. Without this mapping, an exhaustive search for the index would be required.

Maintaining a sorted list is not reasonable because a large amount of inserts and removes are performed on this list. The additional sort during the application of the core operations is too expensive even if a distribution sort is used. With n surfels, the search for an index is in $O(\log n)$, but a removal or insertion of a surfel would require $O(n)$ operations. The linear list of the rendering subsystem requires all surfels, beginning from the insert or delete position, to be shifted.

A cleanup, however, may help to improve the performance of the rendering algorithm, but this has not been verified. As the size is fixed and the indices are unique, a distribution sort mechanism, e.g. a radix sort, is most suited for this task. The required update and sorting time is then in $O(n)$ and circumvents a large impact on the performance. The rendering subsystem does not require the index-list to be sorted, and this sorting may be performed in a lazy manner. Only if the degree of fragmentation is large enough, i.e. a certain amount of inserts and deletes have been applied, a sort is performed. As stated before, sorting is only reasonable after evaluation has finished. This avoids additional fragmentation caused by the pending core operations.

When deleting an index in the list, this particular index is swapped with the last index in the list. This effectively reduces its size by one. More efficient is a direct exchange of the indices, e.g. a delete directly followed by an insert. This can be implemented by simply replacing the index at the position, avoiding the copy of the last element. If the index to be deleted is the last one, a swapping is also unnecessary. With both approaches, an insert and a delete is performed in constant ($O(1)$) time as long as the index positions are available in constant time.

After application of the `refine`- and `coarse`-operations, the index-list contains all primitives' indices currently present in the *TreeCut*, and it is made available to the graphics card. The GPU renders all surfels identified by this index-list. We choose the index-list approach to select the surfels instead of displaying the complete VBO in the graphics card, such as proposed by Carmona and Froehlich [CF11].

To avoid a dynamic expansion of the buffer that holds the indices – during run-time –, an initial size needs to be defined. This size is chosen by the programmer and should, ideally, be large enough to hold the complete 3d data in finest detail if possible. This way no resize during rendering will occur. If the limitation of the system restricts the surfel count to be drawn, the index-list's size, however, will be set to this limit.

Serialization of the *TreeCut*

To save the representation along with the precalculated data onto disk, a serialization is implemented. Both the tree and a *TreeCut* are then generated using this information. Alternatively, it is extracted from various other formats, like the format provided by the QSplat rendering system.

It is not required for the *TreeCut* that the underlying data source provides an accelerated search for nodes. The implementation of an octree for the representation, however, leads to a better serialization of the data because most information can be extracted solely from the hierarchy. It is also possible to reduce the number of stored surfels in the graphics memory by only uploading the current cut. Also,

a traversal of the tree within the GPU could increase the performance of the system similar to the GigaVoxels approach [Cra+09].

If a progressive multi-resolution structure is used, the expansion rules along with the cut distribution are stored instead. The *TreeCut* has no restrictions regarding the serialization technique applied by the data structures.

6.3.2 Rendering with the *TreeCut*

After the data has been uploaded to the graphics card memory, rendering with the *TreeCut* is straightforward. Using the extracted index-list, the rendering subsystem is able to display the data as point primitives[2]. We refer to this approach as Plain splatting. To achieve a high quality rendering, approaches like *Perspective Accurate Splatting* [Zwi+04] and *Phong Splatting* [BSK04] are required for rendering.

The *Deferred Phong Splatting* [Bot+05] (refer to section 'Point-based Rendering Methods' on page 47) requires two rendering passes plus one additional pass rendering to a full-screen quad. The application of deferred rendering allows fast light calculations and has become very popular in recent years. The authors of the *Phong Splatting*-algorithm provide the source code of the vertex and pixel shaders. We expanded the system by several means and implemented them as a CG shader-technique. In the following, this method will be referred to as Phong splatting.

The increase in visual quality using the *Deferred Phong Splatting* method comes with a penalty in speed because two full passes of processing the geometry are required. During the final full-screen quad pass, no geometry processing occurs. A combination with other methods is not trivial due to the accumulation required for the Phong splatting method. A normalization step is performed during the post process, which correctly weights all splats within a scene. This normalization information is stored in the alpha buffer, and therefore this cannot be used by other methods to encode information.

To enable other applications, like the stippling approach (see section 'Stippling Drawing'), the `Draw`-method can be adapted to achieve the desired results. The *TreeCut* is not limited to a single type of representation as long as a cut (and the correlated surfels data) is sufficient for rendering. If a closed surface representation is required, the Cut-Criterion needs to be fulfilled as well.

6.3.3 Multi-Threading

Common rendering subsystems are implemented as state-machines because an event- or message-based system introduces too much overhead and can stall the overall system. Furthermore, if an event that alters the data is processed during rendering, a false display occurs or, even worse, a race condition is generated. As raising events is not controlled by the application, other solutions are intended.

In the case of the *TreeCut*, we have a assured the separation of rendering and representation data. Additionally, the rendering subsystem does not need any data access to the *TreeCut* regarding any node-structure information.

[2] Point primitives are supported by both DirectX™ and openGL.

One option is then to update the *TreeCut* if rendering is not active. The more *TreeCut*s are used in a scene, the more benefit is gained from parallel processing. The rendering subsystem can display other objects while a cut is being evaluated.

Another, more efficient, option is to maintain a copy of an index-list and to iteratively update the *TreeCut* using this list. Once the processing of the cut has been completed and a new representation is available, the index-list is copied and the rendering continues with the most current cut. This way an exclusive access is only required when the index-list is being transferred. In combination with an out-of-core layout of the data, the data on the GPU is handled more efficiently. Both memory access and consumption are reduced. It is important that any *TreeCut*-operations only operates on the index-list provided by the evaluation strategy because otherwise the data for rendering would be corrupted.

In the following, two strategies are presented to apply multi-processing to *TreeCut*s, which operate at different levels. The first is the parallelization of the evaluation and render methods. This is referred to as *local*. The second type is the parallelization of the `Update - Draw` loop, i.e. the call to the evaluation method and the drawing of the current cut. As this layer represents a more overall parallelization of the *TreeCut* methods, this layer is called *global*.

Internal Parallelization

The *local* parallelization directly affects the loops within the evaluators and the rendering. Its application does not prohibit *global* parallelization. In the following, we only theoretically inspect the parallelization of evaluator algorithms. Furthermore, we assume that efficient pipelining of rendering is taken care of by the rendering subsystem.

The traversal as well as the update of the *TreeCut* allow multi-threaded processing. The first is achieved by invoking a thread for each cut-node as each node is independent of the next. Furthermore, the *TreeCut*-operators keep the next and previous nodes with respect to the linked list order.

For the priority evaluation, the traversal is especially important because a partial sorting for lowest as well as highest priorities is needed. The construction and manipulation of the priority queue, however, needs to be thread safe, which may cause performance loss. One possible solution is to use so called critical sections, but a critical section free method will be faster.

Updates to the data structures, which include the change of the index-list entries as well as the *TreeCut*'s cut-nodes, can be performed in separated threads. But, due to the overhead imposed by the generation of separated threads, we assume that this approach does not offer large performance gains for these small, low-level operations. However, the performance of the evaluation and the rendering will increase if the algorithms are well tuned for parallelization in a *local* manner.

Separating Rendering and Evaluation

Global parallelization is best prepared for by incorporating a mutex[3] or semaphore system. Referring to the separated data representation, the exchange of the index-list has to be exclusive. If the rendering

[3] Mutual Exclusion

data in the graphics card memory needs to be altered by any means, the access has to be locked as well. Therefore, the following definition is given:

Definition 4:

As long as the evaluation method of the cut does not alter the data required for rendering, it can be processed in parallel to the `Draw`-routine in a *global* manner.

Applying this definition, for example, to the bucket strategy, the target bucket calculation and changing the cut representation is performed in parallel. The exchange of the index-list must be done using an exclusive lock as the list is required for rendering.

The optimization algorithm has similar preliminaries regarding changes made to the rendering data. The generation of the priority queues includes a partial sorting of nodes as well as the process of identifying whether a node is a `coarse-parent` or not. All preliminaries can be checked in parallel to drawing of the cut. As the rendering data is separated from the tree representation, altering the *TreeCut* does not corrupt the visible outcome. Only the exchange of the final rendering data, i.e. the index-list, needs to be locked to avoid race conditions.

If a sorting by priority is not possible, e.g. because the data is not available at the current moment, the overall evaluation has to be delayed until these requirements are met. One could circumvent these restriction by incorporating a rapid expansion, which allows application of the evaluation algorithm without any priority sorting. This can be applied in cases with the condition that the imposed restrictions are not fulfilled. This happens in scenarios where the size of a *TreeCut* is increased or decreased by a large amount. This approach, however, has a drawback. It will lead, with high probability, to an suboptimal solution, and a reordering has to be performed once the priority data is available. One has to decide if such an expansion without any priority information results in a useful representation or if the stalling of the evaluation process does not affect the performance. However, in the results presented in 'Evaluation Performance', the early expansion has not been implemented.

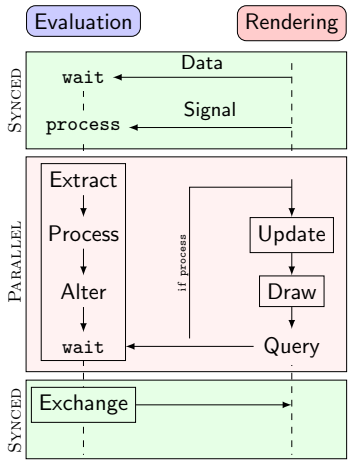


Figure 6.8: The sequence of processing, evaluating and optimizing an existing *TreeCut*. The individual stages encode the availability of data between the render and the evaluation process.

To control the parallel threads in means of exchange and processing the data, a messaging system is proposed. This consists of two stages, namely `wait` and `process`. The communication and execution sequence is shown in figure 6.8. This method is designed to avoid the use of locks during the multi-threaded processing.

If the evaluation thread is in the `wait`-stage, all data has been prepared for evaluating the cut. This means, internal priority queues have been initialized and the cut has been selected, but is not traversed, yet. Also, the index-list has been prepared by the evaluator. Yet, it does not contain the

correct indices.

After this transfer, the evaluation thread is wakened. The stage is set `process`, which informs the *TreeCut* that processing of the data has started. The exchange of the required data is covered by the `Extract`-step. This includes the transfer of the currently used index-list. Also, additional information is generated. In the example of a priority evaluation, the individual priorities of the surfels are fetched. During processing, both the *TreeCut* and the evaluator index-list are updated to represent the changes made by the evaluation method. Note that at this stage the rendering does not reflect this changes because of the data separation performed earlier.

Once the evaluation has finished processing, the stage is set to `wait`, and access to the altered index-list is safe. The rendering thread queries for this stage and swaps the new, altered index-list. After this step, the rendering finally reflects the recent changes made to the *TreeCut*, and the evaluation can be restarted. We additionally query, whether the evaluation has generated a new index-list. This avoids multiple copies of the same index-list as the evaluation has not generated a new representation, yet.

6.4 Results Using the *TreeCut*

Based on a prototype of the *TreeCut* written in `C++`, different scenarios were tested and evaluated against normal LOD-rendering approaches. We evaluate the performance of rendering with the *TreeCut* and the individual evaluation strategies. We also present results generated with the *TreeCut* and possible applications of the proposed evaluation strategies. These results have also been published [SK12a]

When a *TreeCut* is used, it has been generated based on a tree representation of an object. The detail and accuracy of the representation is thereby limited by the parent surfels generation method. Therefore, when a comparison has to be made between the visual results, the same data and method must be used to avoid influence of more accurate generation methods.

The generation method used is the same as in the QSplat algorithm, which calculates the average of the child surfels to determine the position, color, and surface normal. The point-size of each parent surfel is set to cover the whole area of the child points including their point-size, i.e. the Cut-Criterion is fulfilled (see definition 2).

As the tree creation can be performed in a preprocess step, it is omitted in the time measurements. The effective times required for each presented method will be listed in the according subsection. These timings were gathered with the multi-threaded approach presented in 'Multi-Threading'. As stated before, the time needed for evaluation scales directly with the number of cut-nodes. The performance to the rendering is limited by the graphics card and the accompanied calls the the rendering subsystem.

Most of the objects shown (bunny, dragon, lion, and buddha) are freely available at Stanford 3d

Object	System	QSplat Method [ms]	Plain Splatting [ms]	Phong Splatting [ms]
Bunny	1	7.98	0.43	3.71
	2	10.94	0.40	1.31
Dragon	1	192.47	9.10	36.79
	2	290.22	6.10	9.50
Buddha	1	173.54	7.74	31.70
	2	231.83	5.24	8.56
Lion	1	31.68	1.16	6.29
	2	50.09	1.19	2.20

Table 6.1: Comparison of different rendering methods for static representations. As expected, the *TreeCut* is faster than the recursive algorithm because the representation has a GPU-friendly layout. The larger the object, the greater the benefit of the *TreeCut*. The second system profits even more from the faster GPU.

scanning repository[4] in the QSplat format. The stippling objects are created from images provided by Schäfer and Heep [SH10]. Before measurement, all required data is precalculated and made available to the algorithms. Timing results have been averaged over multiple iterations and are taken on the first two test systems presented in appendix 'System Configurations' on page 211.

6.4.1 Rendering Performance

In order to relate the results and the performance of the following sections, only a rendering of the objects is performed in this test. The timing results represent the time needed for a call to the `Update`- and the `Draw`-routine of the *TreeCut* excluding external calls to the user interface and operating system. No *TreeCut*-operations are performed during the tests.

The QSplat rendering method, a Plain splatting, and a Phong splatting are compared to show the overall performance of the *TreeCut* (see table 6.1). The *TreeCut* outperforms the recursive rendering algorithm presented in the QSplat rendering system. This is due to the VBO-based rendering, which avoids transfer of the data to the graphics card. The Phong splatting is slower than the Plain method because multiple render passes are required.

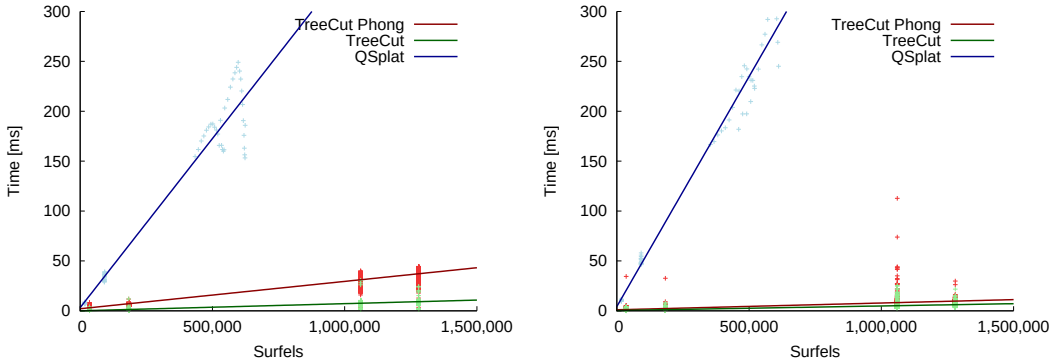
For small objects, the recursive method is faster than the others. However, the larger the object gets, the greater is the advantage of the *TreeCut*. In figure 6.9, a graph visualizes the performance of each method regarding to the surfel count is given.

6.4.2 Evaluation Performance

It is expected that the evaluation algorithms scale linear[5] with the size of the *TreeCut* because all data needs to be processed before an optimization is applied. Also, both core operations require processing time because of the updates to the internal structure and the index-list.

[4] Normal objects are available at `http://graphics.stanford.edu/data/3Dscanrep` and the qsplat objects at `http://graphics.stanford.edu/data/qsplat`

[5] we use a constant partial sorting size, and thus the logarithm in the sort is also constant



(a) Results on Test System 1

(b) Results on Test System 2

Figure 6.9: The ratio between rendering time and surfel count. The larger the surfel count, the more the *TreeCut* outperforms the recursive rendering method.

Several tests are performed that account for various scenarios including large resizes in both directions and complete reconfigurations. Both proposed evaluation algorithms are tested on the same data sets. In the case that the initial size and the target size do not differ, a recombination of the *TreeCut* was forced, so that both `refine` and `coarse`-operations had to be performed simultaneously. This accounts for real world scenarios where large expansions or collapses do not occur when displaying a fixed object.

The expected time complexity for the priority evaluation algorithm is $O(n \log k + k)$ where k is the number of nodes processed per iterations and n is the size of the cut. Here, we assume that both `refine`- and `coarse`-operations are performed with constant time. The $\log k$ factor arises due to the partial sort required by the priority-based selection. The bucket-based approach, however, is more performant since the sorting is not required and the additional application of the temporary results is avoided. This results in a theoretical time complexity of $O(n)$. The last evaluation methods is a copy of the QSplat method. It recursively traverses the tree hierarchy. Due to rejection of splats, a logarithmic time is expected. In worst case, the complete hierarchy needs to be traversed, thus the time complexity is in $O(N)$ with N being the size of the hierarchy.

The graphs in 6.10 show the performance of the evaluator algorithms and includes linearly fitted curves of the times. The dependency of the *TreeCut* size against required processing time is plotted to visualize the time complexity. As expected, the bucket evaluator performs linear with the size and is faster than the priority evaluation. For the latter, the partial sorting of the results has the major performance impact. However, the parameter k of the partial sort is constant throughout the program, and thus a linear dependence on the *TreeCut* size is given as well.

6.4.3 Results Using Priority-based Cut Evaluation

The priority method (see section 'Dynamic Strategies') allows multiple applications due to its universal design. As long as a minimization or maximization problem can be stated, a greedy algorithm

correctly models the required updates to the *TreeCut*. A theoretical algorithm along with the proof of correctness is presented by Carmona and Froehlich [CF11]. As the *TreeCut* uses similar operations, the same findings apply.

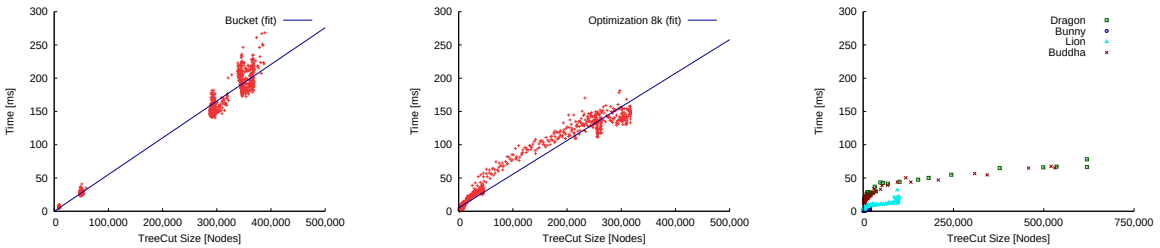
In the following examples, the representation is changed using a priority-based weighting of the nodes. The universal definition of the *TreeCut* also allows application in non 3d rendering scenarios, which is shown in a second example (see section 'Attention Drawing').

Size Restricted Rendering

The priority-based evaluation method requires a criteria to allow smart selection of the nodes. It is theoretically possible to assign the same weight to all nodes, but then *TreeCut* is unable to select the nodes for evaluation correctly and may produce useless results. One should apply the *TreeCut* only when a limitation regarding the size or rendering time is needed. The additional maintenance performed by the *TreeCut* reduces the performance.

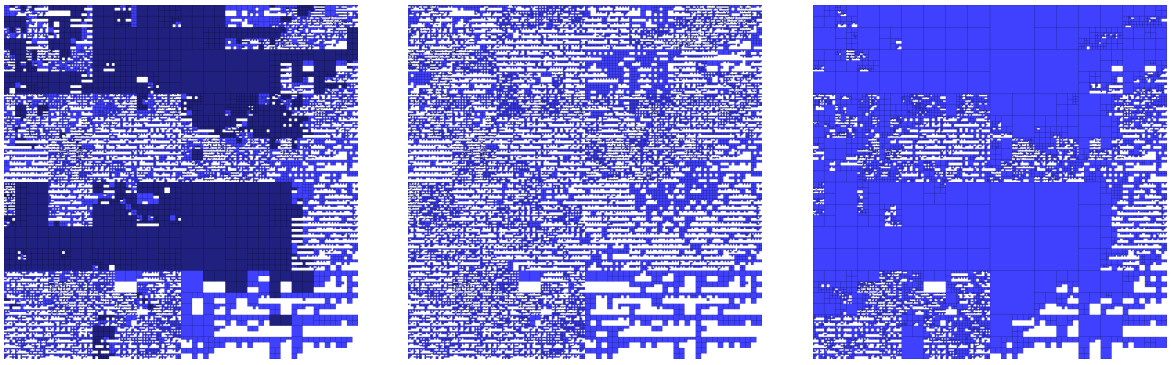
The *TreeCut* may be optimized with respect to different factors, such as cut size or rendering time. If a time criterion is the limiting factor, the size of the cut is adapted until the required time of rendering matches the given time criterion. Within the following example the size is selected as the optimizing factor. The *TreeCut* selects unimportant nodes and reduces them to free memory for more important ones. The surfels with the highest priority values are `refined`, and the surfels with lowest priority values are `coarsened`. These priority values are defined by the normal variation (see section 'Curvature Calculation' on page 37) and have been calculated in advance.

As stated earlier, the QSplat rendering system inherently creates a cut, which is defined by the abort criterion, i.e. the minimal point-size. In figures 6.11a to 6.11c, the QSplat, the normal and the optimized *TreeCut* are visualized and the distribution of the cut-nodes is shown. The smaller the individual nodes are within the quad-tree representation, the deeper they are in the tree representation. A quad-tree is used for visualization and the images generated by each method are presented below.



(a) Bucket evaluator (b) Priority evaluator (c) Recursive evaluator

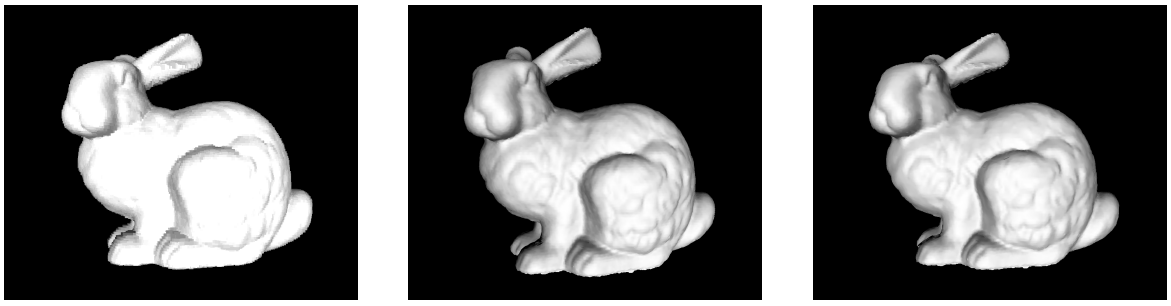
Figure 6.10: The performance graphs of the used evaluator algorithms. The values are given for an average single iteration of each evaluator, and the evaluators have been applied to different objects with varying sizes. The first and the second offer linear time complexity, but the priority evaluation has a larger overhead due to sorting. The recursive strategy utilizes the given hierarchy and offers logarithmic time complexity. Results are taken on test system 1.



(a) QSplat algorithm

(b) *TreeCut* at full detail

(c) *TreeCut* with reduced detail (using the curvature for reduction)



(d) About 20K surfels

(e) About 35K surfels

(f) About 11K surfels

Figure 6.11: Visualization of the different *TreeCut*s created when applying different rendering methods. The top row shows the distribution if the tree is visualized with a quad-tree. The darker quads are rejected due to culling methods. The bottom row shows the according renderings to the presented visualization. In the figures, the used surfel count is given.

In the first quad tree (figure 6.11a), the object is rendered using the QSplat algorithm. Note that the backfacing nodes are rejected by applying various culling methods, which is shown as the darker quads. The second figure shows the same object rendered at full detail utilizing the *TreeCut*. The third (6.11c) applies a reduction of the size to approximately 30% of the leaf nodes. The selection of the nodes is based on the curvature of the individual surfels assigned to each node.

This reduction does not introduce large errors despite the fact that many nodes have been excluded from rendering. The reduction based on the curvature allows to preserve important details in the object, and thus it confirms results of the *Mesh Saliency* approach presented by Lee et al. [LVJ05]. In figure 6.12, the complete saliency metric (see 'Feedback System' on page 125) has been applied to the Stanford lion model. A more qualitative verification is given in section 'Evaluation' on page 135. Additional images generated with the *TreeCut* and its LOD-reduction can be seen in appendix 'Results Using the *Feedback System* ' on page 219.

Attention Drawing

The universal design of the *TreeCut* allows various applications including non 3d scenarios. We have performed a user test to evaluate the relative importance of a notification icon [SSK11]. The icon

(a) HQ Model (183k Surfels) (b) Reduced Version with 80k Surfels (c) Reduced Version with 40k Surfels

Figure 6.12: Results of the size restricted rendering with the Stanford lion. The left images is generated with 183k Surfels while the right version only contains 40k Surfels. Due to the used metric, details have been preserved. This method can only be applied with the priority evaluation method, as a sorting is required.

is alterable with respect to basic saliency features, and the final version of the icon is selected by the *TreeCut*. A weight is assigned to each individual feature, and all possible combinations were created in advance. In figure 6.13, the pseudo-tree along with the tested features is depicted. The final representation form is interactively combined from the *TreeCut* nodes and presented to the user.

The aim is to draw the attention of the user while not disturbing or interrupting her in the current task. In the test, the maximal weight possible for the cut gradually increases over time, which results in a change of the icon. Once the user has noticed the icon, the next part of the test started. In the first part, a user had to perform a visual demanding task, the second was to relax, and the third was mental demanding.

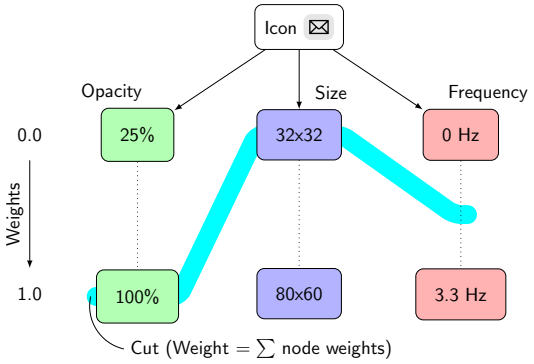


Figure 6.13: The tree used for the attention drawing test. Instead of surfels, icon properties are encoded, and the representation is extracted from the cut.

The results show that the more the user is focused on a task, the higher the influence of the salient features has to be. If a user is busy with a visual or cognitive demanding task, higher average weights have been measured. If the user has been idle, almost no change in weight was required. In figure 6.14, the results of the three performed tests are visualized. The heart rate variability (HRV), NASA TLX, and used time are given in a scale between [0,1]. A time value of 1 corresponds linearly to the maximal time needed to complete a task.

In combination with the results, different tests can be performed regarding the adaptability of the notification icon or the visual salience features. The tests also show that the modeling with the *TreeCut* works well if it is required to increase the visual priority of the notification icon.

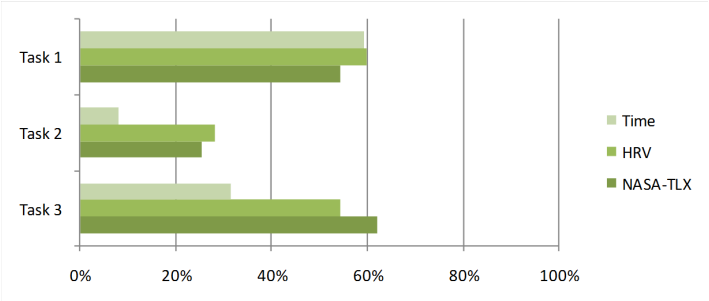


Figure 6.14: Results of the highlight test performed. The participants had to solve three different tasks with various stressful conditions.

6.4.4 Results Using Bucket-based Cut Evaluation

In several occasions, a priority value cannot be determined for a specific set of nodes. In these cases, the bucket-based evaluation method, as described in section 'Dynamic Strategies', provides an alternative processing method. During evaluation of the *TreeCut*, individual nodes are assigned to a bucket, i.e. a level or range of levels within the tree. Based on a difference between the current level and the target bucket of a cut-node, the according operation, i.e. `coarse` and `refine`, is executed.

This method is well suited for a stippling impression of an object, and in combination with the point-based representation of the surface, the density of these surfels create the impression of structure and illuminance on the surface. Stippling images are usually created by artists and have a limited number of predefined views. With the ability generate these images within a computer, the time required for drawing is reduced and customized view can be selected. The dynamic evaluation allows to visualize the 3d object with different light scenarios, so that the structure becomes visible. This way, a more detailed impression of an object is given from the arbitrary views.

For the evaluation of the *TreeCut*, the target bucket has to be selected before the required operation can be performed. In the presented results, equation (6.1) is applied for determination of the target bucket:

$$\text{bucket} = (1 - \max((\vec{N} \cdot \vec{L})^{\gamma}, 0)) \cdot d \tag{6.1}$$

where $\vec{N}$ denotes the surface normal of the current surfel. $\vec{L}$ is the vector towards the position of the light source. By combining the Lambertian illumination $(\vec{N} \cdot \vec{L})$ with the depth of the tree (d), we establish a mapping between the surfel's current level and its bucket. To increase the influence on the illumination coefficient, a non-linear transformation is applied, which is controlled by the variable γ. The bucket can be clamped to a minimum and maximum level to further control the distribution of surfels.

Before an object is drawn with the stippling method, a preprocessing is required, which is covered in the next section. The visualization of the data is performed with a adapted Phong splatting technique. Finally, we present a set of results and give some notes regarding the achieved quality.

Stippling Data

The data for generating the stippling images is extracted by a reconstruction algorithm proposed by Schäfer and Heep [SH10]. The technique uses multiple total focus images, which result in a height-map, e.g. a 2d map encoding the surface structure of an object. The map is sampled at a user defined rate to extract a set of surface points. Currently, only a linear interpolation within the height-map is possible, but the sampling rate may be set without any restriction. This enables us to acquire enough surfels for rendering.

The rendering method requires additional information, i.e. the point-size. This is computed from a NNG, which can be constructed efficiently using a z-order curve (refer to section 'Calculation of Geometric Properties' on page 33). With the help of a NNG, the required point-size is extracted either as a maximal (max) or mean point-size. The first selects the xth-nearest neighbor; the surfel's point-size is then given as the half of the Euclidean distance between the position of the surfel x and the current (source) surfel. For the mean point-size, all point-sizes are averaged. In our tests, the point-size is calculated using the max-method and 5 nearest neighbors.

The tree structure for the *TreeCut* is created in a top-down manner with space-partitioning similar to kd-trees. The properties for a parent surfel, e.g. the point-size and the position, are extracted from a randomly chosen child. This method is chosen to avoid movement of surfels once the evaluation has started and the *TreeCut* is changing the representation. After tree and *TreeCut* generation, the bucket evaluation starts and rendering is performed with a adapted Phong splatting technique.

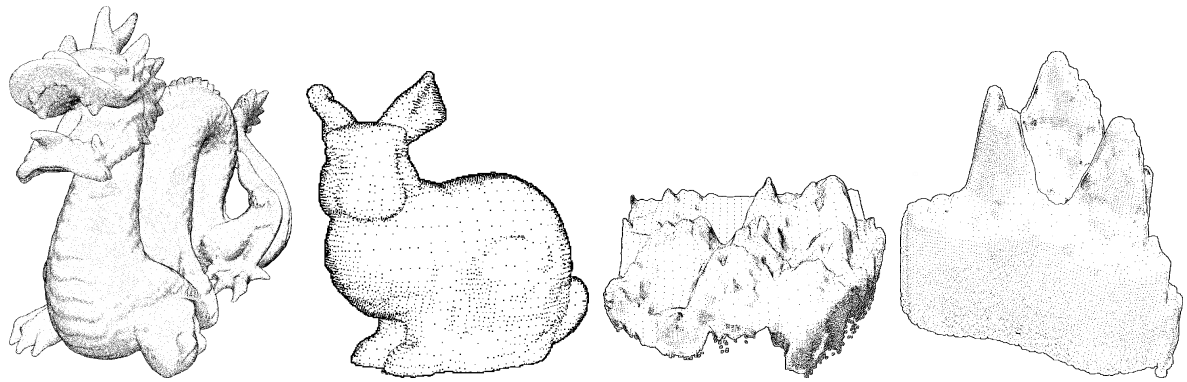
Stippling Drawing

The drawing can be performed with the default *Deferred Phong Splatting* algorithm presented in [Bot+05]. However, to further enhance the quality of the surface representation and the resulting images, some enhancements are proposed.

In the depth pass, all leaf nodes are drawn to recreate the surface instead of the approximation given by the current cut representation. This increases the detail of the reconstructed surface and reduces effects due to approximated surfels in the hierarchy. The surfels are rendered using a dynamic point-size to assure a closed surface. This way, we achieve the best results for removal of backfacing and hidden surfels.

For stippling images, it is common that the points have the same size as well as the same shape. Therefore, surfels are splatted with a user-defined point-size in the attribute pass. This enables the user to change the result by selecting an appropriate point-size by hand.

The last pass enhances the edges of the drawing and the final splats are generated in the frame-buffer. The edge drawing is performed with the help of the image enhancement algorithm presented by Luft et al. [LCD06]. The *Image Enhancement by Unsharp Masking the Depth Buffer* operates on different LODs of the depth-map and enhances differences found in them. This reinforces silhouettes as well as important inner edges of an object. The splats are drawn as proposed by Botsch et al. [Bot+05] as ellipses with perspective correction, but with a fixed point-size.



(a) Stanford dragon (b) Stanford bunny (c) Eochenus (d) Leptictidium tooth

Figure 6.15: Result of the stippling method with various objects. We use both reconstructed and existing 3d objects for display. The *TreeCut* and the evaluation strategies, here the bucket-based approach, are independent of the underlying representation.

Stippling Results

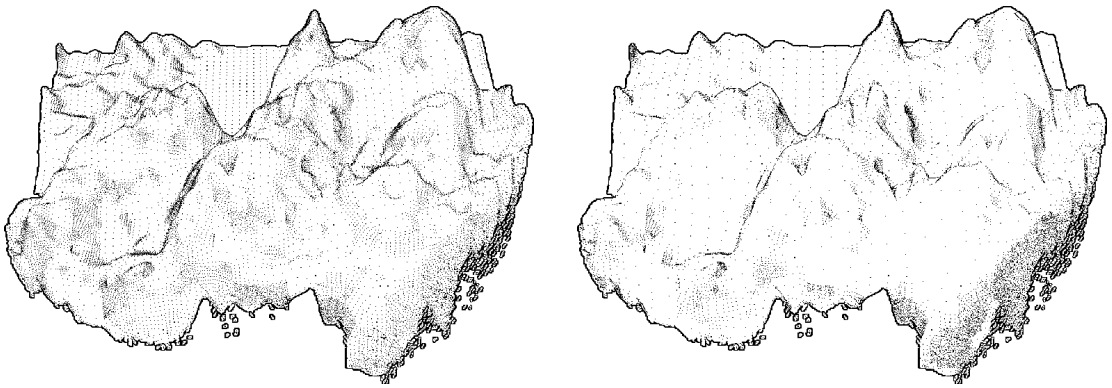
The combination of the bucket-based evaluation and the stippling drawing method creates good results. Interactive frame rates are achieved on test systems 1 and 2 (see appendix 'System Configurations' on page 211). The evaluation is performed in parallel, therefore circumventing a stalling of the system if the bucket algorithm requires more time for processing.

In figures 6.15a to 6.15d, several results generated with the presented algorithm are shown. In figures 6.16a and 6.16b, the influence of changes is shown caused by the additional variables of formula (6.1). The first has a lower overall setting (minimum, maximum level) while the second presents the effect of altered illumination by changing the γ-value. Additional results using the stippling approach of the *TreeCut* are shown in the appendix 'Results Using the *Feedback System* ' on page 219.

This approach inherently offers frame coherency regarding the surfels' positions which cannot easily be achieved by post-processing methods commonly used. Additionally, the current approach avoids the so called shower-door effect where the stippling pattern appears like a curtain laid over an object. This effect usually occurs because the pattern is a simple texture applied to the final frame.

The quality of the results is good, but the tree hierarchy does not yet inherit important properties, such as Blue Noise [HDK01]. Blue Noise could be preserved, for example, during sampling of the height-map or during the hierarchy generation.

Another drawback arises from the fact that hierarchy generation is performed in a top-down manner, which results in different depths for each subtree. The proposed algorithm selects the maximal depth of a tree, which does not produce good results. For example, this can be seen at the back of the dragon in figure 6.15a. A large cloud of surfels is generated, which is unwanted. An ad hoc solution would be to provide the real subtree depth during bucket calculation. This, however, has not been tested and may interfere the mapping of the levels.



(a) Eochenus tooth with max level 8, min level 6 and $\gamma = 3.9$

(b) Variation with max level 9, min level 5, $\gamma = 2.1$

Figure 6.16: Influence of the parameters available in the bucket calculation of the evaluator.

6.5 Conclusion

The *TreeCut* incorporates multiple important properties for perception-based rendering and allows interactive rendering depending only on the system capabilities. The *TreeCut* can be limited to a maximum capacity or a target frame rate, and it is independent of the rendering subsystem.

A major feature of the *TreeCut* is that it retains the representation of the object and allows manipulation using two simple operations. Thus, no retraversal or reevaluation of the object is required for redrawing.

This allows to dynamically change the LOD of an object without having to declare predefined LODs in advance. The dynamic update enables us to define different evaluation methods, which allow the cut to adapt to the environment as well as external limitations. Two strategies were presented, one being an optimization based on the properties of the current cut, and the other is a bucket-based approach. The first uses a greedy algorithm that selects the best distribution of the cut-nodes. The second assigns a bucket to each cut-node to choose whether it must be `refined` or `coarsened`. This approach has lower computational requirements because no sorting is required.

Furthermore, we have separated the evaluation from the *TreeCut*. Thus, it allows parallel processing. Only during the exchange of the rendering data, a lock has to be established. The application controls when this exchange will occur and is able to reduce the time penalty when transferring data between the two threads.

The *TreeCut* is implemented within an MVC-pattern, and thus it can be integrated into existing applications. The rendering system does not have to account for the extensions made available by the *TreeCut*.

However, further improvements of the presented system are possible. To allow smooth transitions between the levels in the data structure, a more general representation of the cut has to be defined. It would allow to reduce flicker artifacts that appear during transitions. This may be especially interesting for perception-based rendering applications as flicker or discontinuities are a strong perceptual

features.

The multi-threading aspects are basic and are designed for a single object only. We state that the selection of the ideal point for exchange of an index-list should be managed by the *Controller*. This could compare rendering and evaluation times as well as other aspects, such as scene information. When it could be assured that the exchange of the data occurs in parallel to user input or scene recalculations, a minimal performance loss is achieved.

The *TreeCut* could also be implemented within a client-server application based on a progressive data representation. Only a small amount of data would have to be transferred to the client machine, and rendering could be optimized depending on the clients capabilities. The evaluation could be performed on the server, reducing computational requirements on the client-side.

7. Bidirectional Saliency Weight Distribution Function

To allow a rendering system to adapt to a perceptual influence, a rating of this influence is required. The visual salience model offers a low-level identification of regions of interest, but existing models mainly focus on image-based calculations. A partial definition has been proposed, but the derivation of a complete 3d visual salience model will be attempted in this thesis.

As presented before in the section 'Perception-based Rendering' on page 51, several methods exist to incorporate perceptual information into 3d rendering, but most visual salience-based models are screen related and not object-based. In combination with the *TreeCut* data structure (see 'TreeCut' on page 73), these restrictions will be removed, and a more universal representation is proposed.

For this thesis, we adopt the saliency representation of Koch and Ullman [KU85], which is a feature representation to model the early vision in the HVS (refer to 'Preattentive Features and Saliency' on page 21). In the application framework presented by Itti et al. [IKN98], which is based on this saliency representation, an image-based processing is presented. We enhance this method and create a 3d representation of saliency.

Saliency defines how an object or element differs in some form from its surrounding. This could either be a red dot in the center of a white sheet of paper as well as a skyscraper within a suburb town. These objects are processed in a model for visual attention. This results in several features, which have to be accounted for. These features are then combined to create an overall saliency map, which encodes positions of interesting points.

In our case of 3d visual salience, we define an object as the complete representation built up from individual primitives, i.e. the vertices. In other words, an object consists of multiple primitives.

In this chapter, the features to create a saliency representation will be examined and applied to a 3d scenario to develop a universal calculation function for 3d visual salience. This universal function offers several advantages, e.g. storing features in a lookup table, which is presented in this chapter as well.

7.1 Definition

As a single object is considered salient if it differs in some amount from its surrounding, a center-surround mechanism is implemented, which extracts information about neighbors with respect to object in question. The neighborhood, however, depends on the distance of the spectator to the object and has to be taken into account as well. The size of the neighborhood arises from limitations in acuity of the HVS. As a rule of thumb, a human can only discriminate 1 arc second, which is about 1.75 mm at a distance of 6 meters. If objects move closer to the spectator, more details will be seen, but details

will become unnoticeable when the objects moves further away.

Secondly, the sources that form the final saliency value must be determined. Within literature, these sources are named features and represent the individual processes of human vision. For a more extended explanation of these features, refer to section 'Preattentive Features and Saliency' on page 21. These features, which are defined for 2d images, will be converted to their equivalents within 3d space in section 'Mapping of 2d Features'.

The visual salience methods presented in section 'Perception-based Rendering' on page 51 indicate that an acceleration of 3d rendering is achieved solely with bottom-up representations, such as saliency. However, no universal calculation method and no complete representation for 3d visual salience exist. In the 3d case, saliency depends mainly on two factors, and therefore the following definitions are given:

- The camera position that defines how an object will be seen. If an part is not seen, no salience calculation is required.
- The light position that defines what influence an object has to the final image. When an part or an object is unlit, no saliency calculation is required while partial illumination decreases final salience values.

A small example will explain these definitions. Given a scene that are fully lit, i.e. all objects can be seen, with highly salient regions. Now imagine, the light only illuminates some part of the scene while others are hidden in the shadow. Only the illuminated parts will now contribute to the overall saliency information as the other parts cannot be seen by a human spectator. This also holds for partly lit regions because the illumination influences the emitted colors. As humans mainly perceive color information, these regions have a reduced influence on the overall saliency information.

A universal representation needs to account for the available features when calculating the saliency values of an object in question. The Bidirectional Saliency Weight Distribution Function (BSWDF) formalizes the computation while being independent of the provided data and the feature extraction.

7.2 Mapping of 2d Features

Yet, saliency feature extractors are only defined for 2d images and a direct mapping to 3d objects is needed for our 3d visual salience model. In table 7.1, a list with the most important salient features is given. In the following, these features will be mapped to their equivalent in 3d space.

A direct mapping of the orientation features was presented by Lee et al. [LVJ05], who define the surface **Curvature** as an equivalent feature for 3d objects. Other features are converted in a similar manner, and are mapped onto the surface of an object. Additionally, the **Silhouette** is introduced, so that the objects boundaries can be weighted separately.

The Color Differences account for plain surface features that do not include illumination changes as these are covered already by the Orientation. So, the Color Differences of the surface are extracted

Name	Description
Orientation	Determines object boundaries and inner surface changes
Color Differences (R-G, B-Y)	Defines the response of the receptive field regarding the signals from the different cone types
Luminance	The overall incoming light
Motion	Local or global movement of a pixel
Flicker	Visibility of a pixel over time

Table 7.1: A partial list of features used in saliency calculations. Usually color or pixel attributes are extracted as temporal or spatial information is harder to retrieve.

from an albedo map, which does not include lighting information. The luminance feature is accounted for in the same manner. We name these features **Albedo Color Differences** and **Albedo Luminance**.

Object Motion enhances the feature values while limiting the maximum value for the whole object. This is due to the fact that human vision is attracted by movable objects, but cannot focus on fast moving objects correctly. This is due to the temporal acuity mentioned in 'Visual Perception and the Human Visual System' on page 11 (refer to Yee et al. [YPG01]). As a rule of thumb, objects as well as regions that translate differently catches one's attention. Motion is extracted from available position information which is altered by an external source, e.g. a physics library. The relative change to the previous frame identifies the magnitude of this feature. The **Object Motion** accumulates the motion performed by all primitives of an object. It determines the maximum value and the increase of the saliency values. The movement of single primitives is accounted for by **Relative Motion**. The latter only changes, however, when simulating liquids, elastoplastics or soft-bodies.

Flicker is mapped to both **Object Visibility** and **Local Visibility** features. If a primitive or object was not drawn in the last frame and is drawn now, its saliency value is increased for a short time period. This value decreases over time as long as it remains visible. As with the **Object Motion** feature, **Object Visibility** is applied to the whole object instead.

A summary of all proposed 3d visual salience features is given in table 7.2. Additionally, a short description of each feature is given.

7.3 The BSWDF

Our universal calculation function for visual salience depends on two input parameters. This is similar to a Bidirectional Reflectance Distribution Function (BRDF), which models the reflectance behavior of a surface in dependency of both light and camera position. These positions are given in spherical coordinates and take values in a hemi-sphere around the evaluated surface point. In our scenario, a full sphere is required to account for all positions of both an object and the light.

However, sphere coordinates do not include the distance of the spectator to the surfel for which the saliency information has to be calculated. Therefore, we have to provide the distance as an external

Name	Description
Curvature	Accounts for changes in the surface of the object.
Silhouette	The object boundaries depending on the current viewing angle
Albedo Color Differences	The surface color differences independent of the light information The value results from primitive or texture colors
Albedo Luminance	The surface illuminance extracted from the same color values as above
Object Motion	The motion performed by the object regarding the last simulation step
Relative Motion	Motion of a single primitive with respect to the last simulation step
Object Visibility	The global visibility of the object Accounts for flicker effects at the object level
Local Visibility	Determines whether this primitive has changed its visibility regarding the last frame Accounts for flicker effects at the local level

Table 7.2: Proposed list of features for 3d saliency calculations. The selection of the features is inspired by the features extracted from 2d images. In this case, we directly refer to 3d objects and primitives because the feature extractors are designed to operate on them.

parameter during computation. The surface position is identified by a vector pointing to the surfel. In the following, we assume that a surfel is represented by a vertex, but it could also be the face of a polygon.

The full sphere and the additional parameters define the scenario in which the BSWDF is calculated, and it is depicted in figure 7.1. In the figure, the requirement for light and position becomes evident because a surfel might not be visible from the camera's point of view or may be unlit due to the light's position. In both cases, the evaluated surfel would not contribute to saliency information in this configuration.

Definition 5 (BSWDF)**:**

Let ω_C be the position of the camera and ω_L the light in spherical coordinates and d_C the distance of the camera to the surfel $\vec{x}$. Then, the Bidirectional Saliency Weight Distribution Function (BSWDF) is defined as:

$$\text{BSWDF}(\omega_C, \omega_L, d_C, \vec{x}) = \text{Illu}(\omega_L, \vec{x}) \circ \text{Features}(\omega_C, d_C, \vec{x}) \tag{7.1}$$

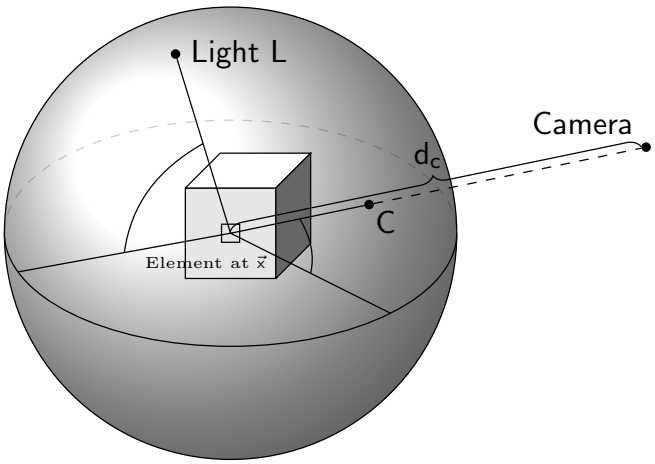


Figure 7.1: The scenario for calculation of the saliency information. The input parameters regarding camera position ω_C, camera distance d_C, light position ω_L as well as the position of the current element $\vec{x}$ are provided to the BSWDF.

The BSWDF comprises two sub methods, which extract information for illumination and features. These methods are variable and have to be replaced by a function that evaluates the needed portions. In our use case, calculation of illumination is implemented as a simple lambert formula, but any illumination model may be used, e.g. a full spectral-based global illumination. However, these models need to correctly scale the individual features, e.g. the color differences. This is represented with the $\circ$-operation in the BSWDF. The feature extractor accounts for surface features (see section 'Mapping of 2d Features'), and it combines them into a format that is compatible to the input of the illumination function.

The BSWDF is not only applicable to 3d objects, but to images as well. The dimensionality of the input data requires to use the correct feature extraction methods.

In the case of 2d images, the illumination is omitted as lighting information is already included in the pixels. Also, the camera only encodes distance information. Applying the function to 3d objects, the illumination term evaluates lighting information of the surfel in question. It is important to note that the illumination has to account for the correct inhibition of the available features. The light source may change the emitted color, and thus the way, the features are perceived.

The feature extractor simulates the basic operations performed by the HVS, and it accounts for a variable set of features, e.g. color differences, luminance values, or orientation. The extraction of features uses the mapped operators in the 3d case, but in the 2d case, the original feature extraction methods are utilized. Thus, the features function is defined in dependency of the dimensionality of the input data as follows:

$$\text{Features}(\omega_C, d_C, \vec{x}) = \begin{cases} \text{2d-Features}(\omega_C, d_C, \vec{x}) & \text{for images} \\ \text{3d-Features}(\omega_C, d_C, \vec{x}) & \text{else.} \end{cases} \tag{7.2}$$

In the following subsections, the specialized versions of the feature extractors are presented, and

the type of extracted data is explained.

7.3.1 2d BSWDF Calculation

In the 2d visual salience calculation, images provide the data that will be processed by the feature extractors. The input parameters ω_C and ω_L account for any position changes of camera and light. In the 2d case, these parameters do not hold any valuable information and are discarded in further computations.

The surfel in image space is designated by the $\vec{x}$ parameter and identifies the pixel to be calculated. This value is expected to be within the extends of the provided image.

The distance parameter d_C holds the scale for the center-surround extraction. The greater the distance, the greater the surrounding of the inspected pixel. This is implemented by increasing the neighborhood of the evaluated pixel.

Feature Extractors

The list of the examined features is presented in section 'Mapping of 2d Features', and each extractor assures that the values are normalized. This allows a free combination of the available features without invalidating the data. This normalization includes a (linear) mapping of the values in the predefined range of [0,1] as well as a (non-linear) scaling based on the number of local maxima. In our case, the normalization operator presented by Itti et al. [IKN98] is applied.

The individual weighting of all features is implemented with a vector w, which introduces an additional degree of freedom. Itti [Itt05] states that the features do not contribute equally to the saliency of a scene, but vary with each scenario. For example, if the color differences are more accurate within a scenario, their weight may be increased appropriately while decreasing the weight of the others to better suit the given information. If all weights sum up to 1, an additional linear normalization after combination is not necessary.

Definition 6:

Given a list of 2d operators $O_{2\mathrm{d}}$ and weighting factors w, the feature extraction in 2d space is defined as:

$$\text{2d-Features}(\omega_C, d_C, \vec{x}) = \sum_{\mathrm{op} \in O_{2\mathrm{d}}} w(\mathrm{op}) \cdot \mathrm{op}(\vec{x}, d_C) \tag{7.3}$$

The implemented extractor functions have been proposed by Walther and Koch [WK06], and the main extractors for static images are the color differences calculations, i.e. red-green, blue-yellow differences, and luminance differences of each pixel. Multiple orientation filters extract boundaries and edges that isolate individual objects within a scene. This is also performed in the HVS.

For motion pictures, the preattentive features flicker and motion are extracted not only by infor-

mation present in the 2d plane of the image, but also by acquiring information throughout multiple images over time. This allows an estimation of motion, and flicker for a single pixel is detected as well.

Illumination and Saliency Values

Once the features have been extracted, they are scaled according to the illumination. As no change of the available features is intended in the 2d scenario, only a linear mapping of the overall value would occur. This additional scaling is normally not applied, and therefore the illumination is omitted.

Thus, the BSWDF is essentially the same as the saliency calculation methods presented by Itti et al. [IKN98] or Walther and Koch [WK06].

7.3.2 3d BSWDF Calculation

In the 3d scenario, data is provided by the scene consisting of 3d objects. Thus, the data is calculated in the complete scene, which includes light and camera information.

The camera defines both distance and view of the object. As the set of available features changes when a different view is chosen, this parameter is required for the 3d feature extractors. As with the 2d case, the distance d_C increases the size of the neighborhood of a surfel if the distance to the object increases. The surface position $\vec{x}$ provides access to local surface properties, which are required by the feature extractors as well. ω_L defines the position of the light source that may influence the extracted features for derivation of the final saliency values.

In figure 7.2, a flow chart visualizes the steps performed by the 3d calculation of the visual salience.

Feature Extractors

The feature extractors are adapted as stated in section 'Mapping of 2d Features'. To yield highest efficiency, these features are extracted during rendering, e.g. during the pixel shader-stage (refer to section '3d Rendering' on page 5 for an introduction into the render pipeline and shader terminology).

The center-surround operation and the combination of the individual features have to be applied in a separated processing stage because global information is necessary to normalize the features. This information is not available in a pixel shader.

We have identified two types of features, local and global ones. These do not appear in the 2d scenario because there only object-independent data is available. In the 3d case, however, the connection information of objects is used to discriminate these types. During the feature extraction, the global features have to be calculated separately, and they influence the saliency values of an object. As stated in 'Mapping of 2d Features', a global effect should alter the overall weighting for all feature values with respect to that object.

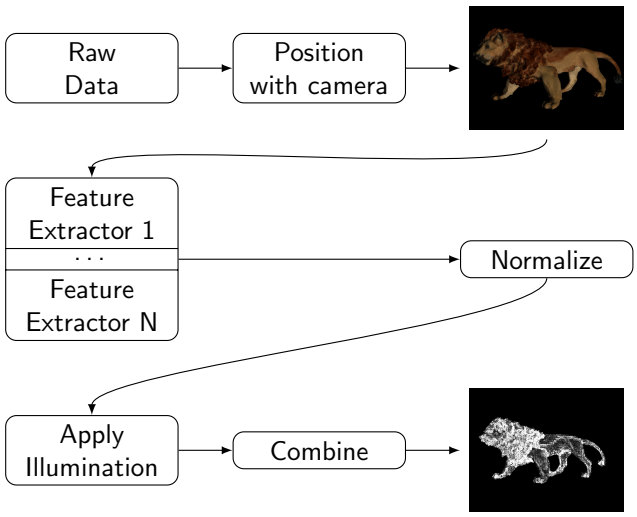


Figure 7.2: Processes in the 3d visual salience calculation using the BSWDF. After positioning the object in 3d space with the given camera configuration, the extractors are applied to the visible surface parts. The normalization allows to combine the features after they have been illuminated. The result is a saliency map of the object.

Definition 7:

Given a list of local operators LO_{3d}, global operators GO_{3d} for 3d feature extraction, and weighting factors w. The feature extraction is then defined as follows:

$$\text{3d-Features}(\omega_C, d_C, \vec{x}) = \left(\sum_{\text{op} \in GO_{3d}} w(\text{op}) \cdot \text{op}(\omega_C, d_C) \right) \otimes \left(\sum_{\text{op} \in LO_{3d}} w(\text{op}) \cdot \text{op}(\omega_C, d_C, \vec{x}) \right) \tag{7.4}$$

In equation (7.4), the global operators do not need the local information, i.e. $\vec{x}$, as the whole object is evaluated in these functions. However, the global operators can influence the outcome of the local operators, e.g. a maximal or minimal value. This is expressed with the $\otimes$-operator.

Opposed to the 2d operators presented in the image-based saliency calculation, the normalization operator must not be applied to all features equally because it could remove important features. This is especially true when applying a curvature extraction that also provides a silhouette. The silhouette has large amount of local maxima, and a normalization would result in mapping most of the curvature features near to zero within the combined feature map. Thus, the curvature and silhouette information would not contribute in the final saliency values. For this reason, a separation of inner curvature features and silhouette needs to be applied. Then, the curvature and the color features are processed normally with the normalization operator.

To account for moving or flickering objects, 3d data is directly accessed and propagated to the feature extractors. To derive the relative motion of a surfel, a history of positions has to be maintained

if this information is not available. The flicker can be incorporated with occlusion culling techniques. Alternatively, this can be achieved by marking clipped or culled surfels as well. Once the surfels are drawn, the mark is removed, but the surfel is treated as a flicker surfel.

Note that the features are extracted for a single object at once, and they need to be combined during rasterization in the z-buffer to result in a saliency map for the entire scene. As each feature extraction renders its information into the buffer, an occlusion is performed automatically. In the final saliency map, object-specific information is accumulated. The complete map identifies regions or areas of interest on the surface of all shown objects.

Illumination and Saliency Calculation

Once the extraction has been performed, the single features are summed up to create the local saliency information for an individual surfel.

For the illumination term of the BSWDF in equation (7.1), the surface normal at position $\vec{x}$ is needed. The illumination model has to include possible inhibition and changes due to the light color. Therefore, existing models have to be adapted. For ease of implementation, a Lambertian illumination is used in the prototype, only linearly scaling the features' values.

To allow efficient and parallel generation of the saliency map, the use of multiple render targets is reasonable, which allows one to create more than four features at once. For processing, a floating point representation of the features is required to obtain reasonable values during normalization.

7.4 Implementation

The BSWDF definition allows an implementation either on CPU or GPU. In the following subsections, the needed operations and steps for an implementation are given. Our CPU implementation is based on the Saliency Toolbox from [WK06], which is freely available[1]. Their source code is written in Matlab along with some `C++`-functions and classes to speedup image processing. The toolbox is essentially based on the framework defined by Itti et al. [IKN98]. We, however, tune our implementation to ideally fit to the rendering pipeline.

The proposed GPU implementation is inspired by the CPU method, but offers some speedups. It utilizes the vector-based GPU architecture to extract the features and increase performance of other operations. The presented methods are different to existing saliency method presented by [Xu+09; WK06]. Both of our proposed methods focus on the evaluation of a scene given in 3d. The CPU method evaluates an image given from the rendering pipeline while the GPU method is integrated in the rendering pipeline, and it directly operates on the geometry data.

As we have generated features for a single object, we are able to reduce the size of the feature maps to the displayed area of the object. An individual feature map only needs to cover the visible parts, and thus a mipmapping approach can be applied. We restrict the extends of a map to the projected

[1] See `http://www.saliencytoolbox.net`

size of the object's Axis Aligned Bounding Box (AABB). This results in a reduction of the size when the features are calculated at a higher distance or for smaller objects. Since the extraction is applied to the visible parts of the object, no information will be omitted, and memory is saved.

7.4.1 Calculation on the CPU

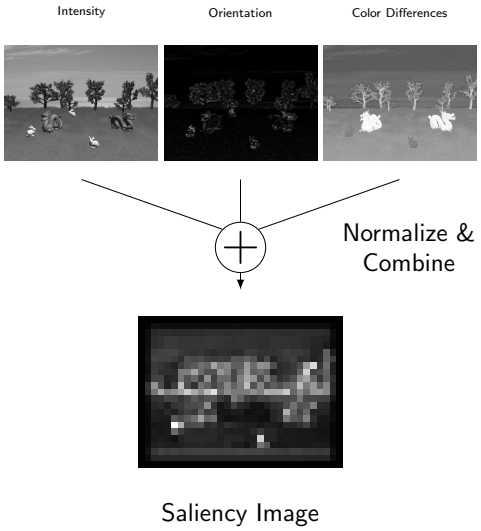


Figure 7.3: Outline of the saliency calculation method for 2d images. The feature maps are extracted from a color image and is combined to result in the saliency image.

The floating point representation of the provided image is processed with a given set of extractors. An outline of the calculation of complete saliency image, often referred to as a saliency map, is shown in figure 7.3. In the following, the extraction of color differences is explained, but the steps remain the same for all other extraction methods. An exception is given in the case of temporal extractors that additionally require a series of images, such as motion.

The given image is reduced in size using a Laplace-pyramid approach effectively reducing both width and height by factor 2 in each mipmap level. Each of these levels is created using convolution with a separated 3×1 Gaussian filter followed by a selection of every second pixel to define the next level image. Within the Saliency Toolkit [WK06] this operation is called `makeDyadicPyramid` because there is a dyadic (binary) relation between the single levels of the images. After reduction, the color differences are calculated for each pixel at each level.

The center-surround method is applied by selecting a main level, which contains the center pixel, and a delta level, which sums the surrounding pixels. The image of the delta level is resized to the extends of the center image by using pixel-repetition. This means that a nearest neighbor interpolation is used. With this operation, the center-surround operation is reduced to a simple pixel subtraction between the level- and delta level-image. This step is done for all feature images available at the desired delta levels. The center-surround operation applies the DOG method as a Laplace-pyramid representation is used. As an alternative to the pyramid representation, one could also increase the number of neighboring pixels instead, but we follow the implementation of Walther and Koch [WK06]. After this step, the individual result images are called feature maps.

Before normalization, an attenuation of the feature maps near their borders is applied. This assures the combination of images will not introduce any artifacts at the edges due to the image-size reduction performed earlier. This also enforces features near the center, which is applied because spectators are expected to look at the center of the image first.

To combine the individual features to the so called conspicuity maps, several iterations of a nor-

```
void normalizeImage(float *imageData
                    , int size
                    , float min
                    , float max)
{
 //clamp data to range [min,+inf]
 //also extract maximal and minimal values
 [lMin,lMax] = clampDataTo(imageData,size,min);
 //using local min and max values to normalize
 normalizeDataInRange(imagedata,size,min,max,lMin,lMax);
 //extract local maxima data
 [maxCount,maxSum] = getLocalMaximaInformation(imageData,size);
 //if we have found at least one local maxima
 //scale the whole data based on the local maxima information
 if(localMax > 1) {
  //calculate scale
  float scale = pow(maxValue - maxSum / (float) maxCount),2);
  //apply scale
  for(int i=0;i<size;i++) {
   data[i] *= scale;
  }
 }
}
```

Listing 7.1: Abstract normalization using the non-iterative method proposed by Itti et al. [IKN98]. The data is clamped and linear normalized before the non-linear scaling is applied.

malization procedure are required. In the first iteration, all feature maps are normalized. Here, for example, red-green and blue-yellow color differences are processed independently. The feature maps at the lowest level are combined by pixelwise addition. These combined feature maps are repeatedly normalized until the individual conspicuity maps are generated, i.e. no maps of a conspicuity type, such as color, are left.

A pseudocode performing the normalization is given in listing 7.1. In first step, all values are clamped to a minimal value. In our case, it is always zero, and after processing the global maxima and minima are returned. These are used to normalize the image data within the range of [minValue,maxValue] with a linear mapping. Afterwards, the local maximas are extracted, which are defined in the Saliency Toolkit as searching the neighboring pixels in horizontal and vertical direction. If the given pixel value is greater than a globally set threshold and its value is greater than the neighboring pixels, the `maxSum` is increased by the current pixel's value, and the `maxCount` is increased by one. When the local minima and maxima have been extracted, the non-linear normalization is applied to each pixel. The more local maximas are found, the more the values will be scaled towards zero. If only a small number of maxima exists, the feature map will not be altered too much. This operator has been defined because of the intuition that a large number of maximas will not provide any indication for an interesting feature [IKN98].

With weighting factors and the individual conspicuity maps, the saliency map is formed. In the case that a visualization is required, another normalization step is useful for appropriate scaling of

the saliency values. If the data will be processed further, this step may be omitted as long as no combination with other features will be performed.

This map is then used either as means to drive a neuronal network for selecting the most salient regions or to process data in a perceptual manner. The first has been implemented by Itti et al. [IKN98] to simulate human fixation behavior while the latter is most suitable for the application with a dynamic data structure, such as the *TreeCut*.

Generally, 3d objects cannot be represented in a single image, so other approaches are required to apply saliency calculation directly to 3d objects. Even if the object is a two-manifold and the representation is given in form of a geometry image [GGH02], the application would not lead to correct results because neighboring surfels are not required to be neighbors in the image representation. Therefore, we propose another approach, and we integrate the saliency calculation into the rendering pipeline.

7.4.2 Calculation on the GPU

The GPU methods enables us not only to reproduce the results of the CPU method, but also allows to implement the calculation during rendering.

A conversion of the 2d saliency map generation on the GPU has been presented in [Xu+09]. A CUDA implementation for the individual filters as well as some adaptations to the individual operators have been proposed, and a massive parallel computation using several graphics cards for the calculation is presented. This method enables to process large data set fast, but still this method does not account for any 3d specific features. Only the 2d operators were mapped onto the graphics card. Yet, it shows that a great performance increase is achieved when using the GPU for calculation because most operations can be performed in parallel. These also benefit from the vector architecture of the GPU. Our proposed method uses a DOG approach for center-surround inhibition, and several performance enhancements are applied because a 3d rendering system is available.

Direct Optimizations

We propose to generate the different Laplace-pyramid levels with help of the GPU or rendering subsystem methods. This performs better than any CPU-based implementation because the algorithm is parallelized efficiently. An according call to the rendering subsystem automatically creates all mipmap levels for a given image, i.e. a texture.

The DOG-method for center-surround of the features selectively compares individual mipmap levels of the provided input image. This allows us to omit unneeded levels, which will neither be generated nor stored in memory. This results in an improvement of computation performance.

Generate Features with Splatting

Such a straightforward implementation is enhanced even further when exploiting the capabilities of the rendering pipeline. Inspired by the point-based rendering, i.e. the Phong splatting method described earlier, some special properties of the rendering can be leveraged for feature calculations. The

extraction of the features is performed in the attribute pass (refer to section ‘Point-based Rendering Methods’ on page 47) to derive the surface features as described in section ’Mapping of 2d Features’. By defining a shader-program that computes the needed information, the features are generated and stored within a single pass utilizing multiple render targets. Instead of an application of an illumination in the deferred pass, the feature extraction is finalized, and the center-surround information is evaluated [SK11].

The DOG approach extracts pixel from different mipmap levels of the provided features, which were generated in the attribute pass. The individual conspicuities are stored in separate RGBA color channels of the result image and form a Layered Conspicuity Map (LCM). The saliency image is generated with the help of the BSWDF that adds light information and weight of the LCMs. Thus, the available information is object-based and solely operates on the 3d data of an object.

The center-surround operation requires the distance information provided by the BSWDF parameters. Due to the accumulation of the individual splats, a smoothing of the result values is achieved, and the center-surround process does not need to be adapted. This allows us to develop and utilize a high-performance implementation for both the complete splatting algorithm and center-surround calculation. In the test cases, a fixed delta level is used for the surround definition.

To explain why the accumulation smoothens the internal features, an example is given. An object drawn at normal resolution does not include any overlapping surfel. The dynamic point-size also supports one by closing such gaps. But at a certain distance, the surfels will start to overlap and overdrawing occurs. Additionally, the dynamic point-size can be used to control the distance when this overdraw will first appear. As the splats are accumulated in the attribute pass, the features will also be summed up. The final normalization, which counts the number of overdrawn elements, smoothens the result data and thus also the individual surface features.

The feature extraction should not be interpolated because it is a per-pixel operation, and artifacts may be introduced when using filtered texture lookups. The individual features are therefore calculated with an unfiltered texture lookup, which avoids any possible interpolation. Thus, the extracted values directly map to the data of input texture given to the feature extractor. The access to individual texture elements (texels) depends on the extends of the texture. These extends have to be provided to the shader-program accordingly. This behavior is different to normal texture lookup methods that use texels in the range of [0,1] for texture access.

To enhance features during difference calculation, an exponential function could be used to increase resulting values similar to a gamma value. However, one has to assure that the values are finally clamped into the range of [0,1].

The curvature of an object can be calculated with different methods. Both precalculation and an in-place calculation, e.g. with our normal variation (refer to section ‘Curvature Calculation’ on page 37, is supported. The latter requires that normal information is available. With our proposed normal variation method, the silhouette of an object can be extracted additionally. We therefore define the dot-product at the object boundaries as 1. As stated before, the information has to be stored

```
1 float4 features = tex2Dfetch(data.featureSampler,data.position);
2 float NdL = dot(lightPos,data.normal);
3 return saturate(dot(features.rgb,NdL.xxx));
```

Listing 7.2: CG code for calculating the BSWDF using a given feature and normal map. The dot-product is available as a hardware instruction in the graphics card and therefore efficient.

separated from the normal curvature, so that normalization can be applied safely.

An alternative method to extract the silhouette is to apply the algorithm presented by Luft et al. [LCD06]. This method, for example, is popular in non-photo-realistic rendering and is covered in section 'Stippling Drawing' on page 95. It utilizes the depth map, i.e. the result of the visibility pass in the Phong splatting technique, to generate the silhouette.

Before the features can be passed to the BSWDF calculation, a normalization is required, so that the individual features become comparable. At this point, these features are referred to as conspicuities.

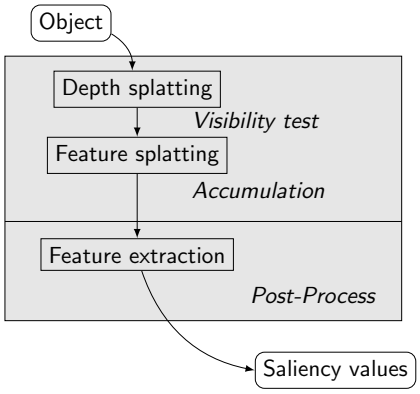


Figure 7.4: The outline of the splatting-based feature extraction method presented in [SK11].

This method is not restricted to point-based rendering methods because the accumulation can be applied to any primitive type. It is advantageous if the overdraw can be influenced, e.g. by altering the primitive's area. In our tests, however, an additional increase of the area was not necessary. With this GPU-approach, the surface features are calculated by a single, well optimized shader. The scale is accounted for inherently by the accumulation method and the distance between object and camera. The features are smoothened appropriately, and these are extracted with a little overhead of normalizing the values prior to processing.

An outline of the presented algorithm is given in figure 7.4. It depicts the individual steps performed during each render pass. The feature extraction is performed in the post-process utilizing the pixel shader, which accounts for correct DOG processing of the data.

Compute Saliency Values with BSWDF

The BSWDF calculation is implemented efficiently in the GPU with a single texture lookup and two dot-products for each vertex. A pseudocode operating on a precalculated LCM and a normal map is shown in listing 7.2. In the code, a Lambert illumination model is applied, but the BSWDF is not restricted to such simple illumination models. However, an illumination model has to be adapted, which is not covered within this thesis and remains future work.

Due to the design of the BSWDF, a separation of feature extraction and saliency calculation is generated. Thus, it is possible to save the created images onto disk for lookup. This is, however, only

reasonable if the data remains static.

7.5 Creating a Lookup

As the on-the-fly calculation of the saliency information does not perform as fast as it would be required for a real-time application, other approaches are needed to accelerate this process. This can be achieved, for example, by updating the available information not every frame or by creating a lookup table for already calculated data to avoid expensive re-computations.

The lookup table idea is valid because the available information varies only slightly in a given scene. Further refining this idea, the recalculation of the saliency information is only necessary if a change of the object has occurred. Think of a static scene in which the spectator is allowed to move freely. Once the camera moves, new information is available, and the data is calculated.

When an object is considered static regarding shape and color, the surface features will not change at all. This allows us to extract these features in advance and provide this information directly to the BSWDF. Therefore, an efficient storage method and a lookup technique is required, so that the data is available for evaluation.

Note that the calculation of the saliency is not limited to static objects, and an approach for a dynamic system is given in chapter 'Perception-influenced Animation' on page 145.

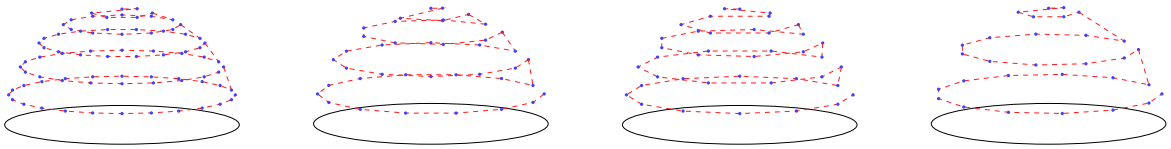
7.5.1 Approach

As with the BRDF functions, one could think of a similar lookup technique for the BSWDF that allows a fast lookup within a precalculated set of saliency maps. Due to the dependency on two factors, i.e. the light and camera positions, the data is stored in dependency of these two factors. But this results in a space requirement of $C \cdot L \leq C^2$ images with C being the number of samples used for the camera positions (including the different distances) and L the number of light positions. This holds as long as $C >= L$. This is the case as we additionally need to store several distances for each camera position to correctly account for the scale of the feature extraction. Furthermore, both L and C cover the same positions which results in $C >= L$.

As opposed to Bidirectional Texturing Function (BTF), a complete-sphere must be covered for both camera and light positions instead of a hemi-sphere. This has a large impact on the memory footprint. An ad-hoc solution is to reduce the number of stored information. However, a reduction of the stored data would remove the ability to freely combine the individual features during final saliency calculation.

The most expensive process is to calculate the features of the object's surface. By storing LCMs instead of saliency maps, the dependency of the light position is removed and one degree of freedom removed (refer to thesis 2). The number of entries in this table is then C instead of C^2 with the additional cost of calculating the BSWDF each time.

Thus, the overall memory requirement decreases dramatically. As the surface normal is available



(a) Original: 81 samples (b) Reduced: 41 samples (c) Variant: 41 samples (d) Compact: 31 samples

Figure 7.5: Different sampling schemes used for creating BTF textures. Derived from the work of Filip et al. [FCH08].

in usual rendering scenarios, e.g. for lighting, the cost for calculating the BSWDF is selecting the appropriate LCM based on the current view of the object, lookup in the texture with possible coordinate transformation (see section 'Lookup The Data'), and calculating two dot-products (refer to listing 7.2). On the GPU, the dot-product is performed in a single instruction, which normally requires a single clock cycle. The other operations can also be performed efficiently on the GPU, and thus is the calculation of the BSWDF performed with almost no cost during rendering of an object.

7.5.2 Sampling Schemes

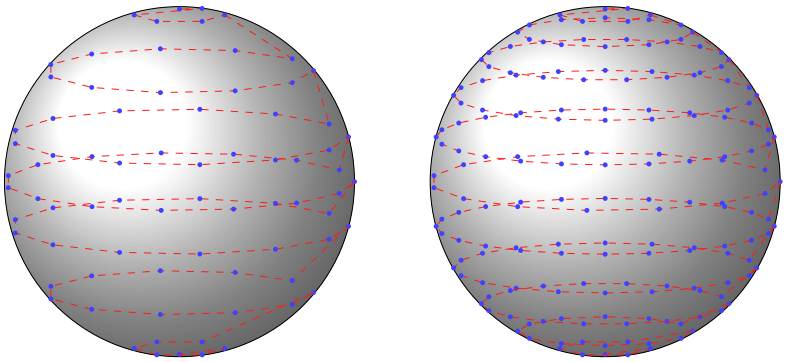
To generate the individual LCMs, an object must be sampled completely. Usually, the sampling positions are defined by a sampling scheme that determines the position of both the light and the camera. Depending on the resolution of the scheme, the number of samples changes, and the greater the sample count, the better the object is represented.

As the sampling is important for BTF-methods, some of them have been tested for accuracy. The derived sampling schemes are based on existing ones. In figures 7.5a to 7.5d, some BTF sampling schemes are shown that were defined by Filip et al. [FCH08]. All schemes try to avoid oversampling of the areas by equally distributing the sampling positions over the hemi-sphere. The schemes begin at the top of the hemi-sphere and circle around towards the equator. Once a ring or stack is completed, the elevation angle is decreased, the azimuth value is reset, and the iteration is started over.

All schemes share the property that an elevation of zero is not used at all. This fact has to be accounted for when expanding any scheme for the sampling of LCMs. Thus, we propose an equally distributed sampling of the azimuth at an elevation of zero. The number of samples is selected, so that it is greater than the samples of the ring created before. In the example of the 77 samples scheme in figure 7.6a, 15 samples are used to create the ring defined at the elevation of zero. The samples of the lower hemi-sphere are mirrored versions of the upper hemi-sphere. The number of total samples can be calculated using this formula:

$$\#\text{samples} = 2 \cdot \#\text{orig} + \#\text{zero}$$

In the case of the 77-sampling scheme, the original sampling scheme has 31 samples and 15 samples for the elevation of zero. This leads to $2 \cdot 31 + 15 = 77$ samples in total. Our naming convention



(a) Scheme with 77 samples (b) Denser scheme with 187 samples

Figure 7.6: Two examples of possible sampling schemes for creating the LCMs. Note that the depth values are not part of the scheme and have to be evaluated on each position.

refers to the number of provided samples.

This number of samples influences the size of the lookup table. As each sample position also includes the distance, each position is sampled multiple times. The distance values are chosen to halve an object in size in each step when using a perspective projection during the rendering. This results in a fixed number of samples for the depth sampling because no reliable features are extractable if an object is at a certain distance, e.g. when the object is mapped to only a few pixels. Furthermore, this halving of the size fits well to the DOG approach as each mipmap level created also represent the object at a doubled distance. The resulting image size is quartered (halved in each dimension) similar to the Laplace-pyramid. We refer to this method as LCM-mipmapping for disambiguation to other mipmap techniques.

The lookup table tests presented are conducted using only one scheme because calculation is independent to the number of samples created. To improve the representation of the features, a denser scheme, e.g. the 187-scheme, might be more appropriate. A comparison between the presented sampling schemes and the achieved visual results should be performed, but is beyond the focus of this work.

7.5.3 Lookup Table Creation

Once a sampling scheme has been selected, an iteration of each sample position is needed. Beginning at the top position the samples are traversed towards the bottom of the unit sphere. At each sample position, the object is moved to the predefined distance values before the next sampling position is selected.

When applying this method, each sample position is represented by different distance values. A linear layout of the lookup table within the memory assures that all LCMs of a sample position reside near to their possible neighbors. This reduces the number cache misses when loading the LCMs onto the graphics card memory or when accessing the different distances in the evaluation of the BSWDF.

```
Scheme entry = GetFirstEntry();
do {
  //set view of object
  setWVPMatrix(entry.GetWVPMatrix());
  //call extraction
  Image *lcm = extractFeatures();
  //store layered conspicuities
  m_lookup.Set(entry,lcm);
  //repeat
} while(entry.GetNext());
```

Listing 7.3: Creation of the lookup table using an iterateable sampling scheme. The `GetNext()`-method of the scheme returns false if the end of the sampling scheme has been reached, true otherwise.

Once the sampling scheme has been selected, the generation of the lookup is pipelined. A pseudo-code illustrates the calculation of the lookup table entry in listing 7.3. In each iteration, the view of the current object is set depending on the current scheme position. The World-View-Projection (WVP) matrix is set accordingly by rotating the object using the azimuth and elevation angles given by the entry. The depth is implemented as a translation along the z-axis based on the near clipping plane of the projection matrix and the extends of the current object. As a perspective matrix is used for projection, the size of the object also depends on the translation part along the z-axis. As presented in section 'Calculation on the GPU', the accumulation method assures correct smoothing of the object's features without further processing. After the position of the object is set, the rendering is performed by applying our GPU method. The features are splatted during the attribute pass of the Phong splatting algorithm and are smoothened in the deferred pass, which also calculates the center-surround information.

The deferred pass assigns the individual features to the color channels, and with the help of render targets, i.e. a framebuffer in OpenGL, these are resolved as an image and can be processed further. By normalizing the features, e.g. with the operator presented by Itti et al. [IKN98], the combination of all features is allowed as stated in section 'Calculation on the CPU'. The resulting LCM is then stored in the memory of the lookup table at the current sampling scheme entry. An example Layered Conspicuity Map Set (LCM-Set) is shown in figure 7.7.

The location of the image in the memory depends on the sampling position provided by the sampling scheme. This also means that a lookup table is bound to a sampling scheme.

Once all positions have been calculated and the corresponding LCM have been extracted, the lookup table can be saved onto disk. As additional information, we store the used scheme, the size of a single LCM and the projection matrix, which is required for the mapping of the stored information. The information is written into the header along with the allocation size of the lookup table. With the number of samples it is possible to load an incomplete lookup table. It is not prohibited by design that new values are inserted into the lookup table during run-time as resolving a lookup texture also grants write access.

Figure 7.7: An example LCM-Set created from the Stanford lion QSplat model [RL00]. The model is resized to fit into a unit sphere and the distances from the camera double each version. The images are cropped from the center to created a mipmapped LCM-Set. We used an elevation of 45 and an azimuth of 130.9 for this view and the LCMs are inverted.

The lookup table is independent of the geometrical resolution of an object. A combination with compression methods allows to store the LCMs with low memory consumption even if an object has a high resolution. The maps can be adjusted in size, so that the density of features matches the quality of the object. We state that the more details are available, the higher the resolution of a LCM should be.

7.5.4 Lookup The Data

To allow a fast access to a LCM during rendering, the derivation of a lookup table needs to be implemented with only a few instructions. Therefore, the following requirements have to be addressed and fulfilled to be suitable within a real-time application:

- The lookup is executable on the graphics card.
- The calculation of the lookup entry does not exceed the normal count of operations performed in an usual rendering system, e.g. it is not a bottleneck of the executing system.
- The lookup allows interpolation of the provided features.

As a first step, the corresponding LCM including all distance versions is extracted. These LCMs will now be referenced as LCM-Set. This extraction is performed by evaluating the camera position. The camera is projected into object space, and the appropriate LCM-Set is selected.

The World-View matrix includes the necessary rotation and translation. We assume here that the same projection matrix as in the lookup table is used. For ease of explanation, furthermore assume that no scaling has been applied to an object and it is visible – if not, no lookup would be necessary at all. By inverting the World-View matrix, the eye coordinates are transferred into object space. This leads to the following mathematical calculation with the camera position (eye) being at $(0,0,0)^T$:

$$\text{eye}_{\text{obj}} = (WV)^{-1} \cdot \text{eye} \tag{7.5}$$

where eye_{obj} is the position of the camera in object space. The multiplication of the camera position with the inverse of the World-View matrix results in the camera position in the object space. An object is expected to be centered around $(0,0,0)^T$. If this is not the case, an AABB is created using the object's extends, and the camera position is adjusted according to the center of the AABB.

The World-View matrix can easily be inverted as no scaling has been applied. The translation is inverted by taking the negative translation values. The rotation is inverted exploiting the orthogonality of a rotation matrix. It implies that the transposed matrix is the same as its inverse:

$$R^{-1} = R^T \tag{7.6}$$

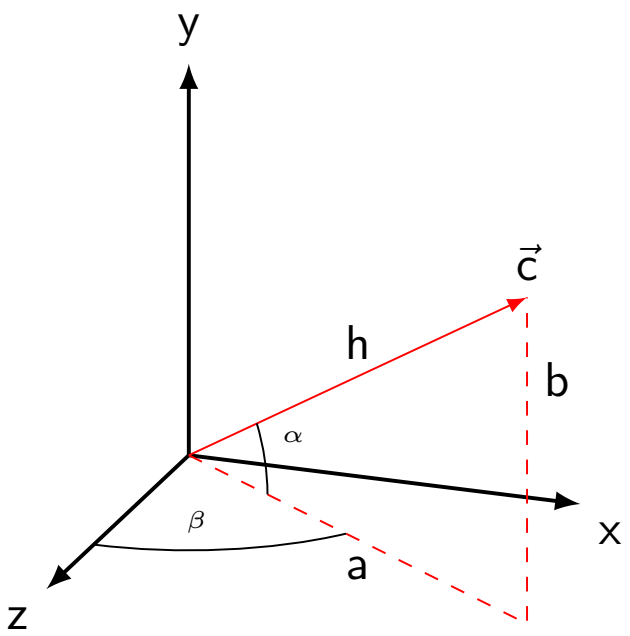


Figure 7.8: Calculation of the lookup table entry using a camera position. The camera position $\vec{c}$ has been projected into object space. Angles α and β will be calculated to derive the lookup entry.

Once the camera coordinates have been transformed, they are converted from Cartesian coordinates to spherical coordinates. The distance value is extracted by calculating the Euclidean distance between the origin and the camera position. For the azimuth and the elevation of the resulting camera position, the scenario is depicted in figure 7.8.

The elevation is defined as the angle between the z-axis and the y-axis on the unit sphere. In the sampling schemes, the coordinate $(0,1,0)^T$ yields an elevation angle of $90°$ while the coordinate $(0,-1,0)^T$ results in an angle of $-90°$. Using the trigonometric functions, the elevation is defined

as the inverse cosine of the adjacent divided by the hypotenuse. The hypotenuse is the distance of the camera position to the origin (h). To avoid projections, we use the y-value of the Cartesian coordinate (b) and calculate $\arccos\left(\frac{b}{h}\right)$ instead. To get the correct elevation, we subtract $\frac{\pi}{2}$ from the result. The azimuth of the sampling scheme is defined in the $x-z$-plane. Thus, the angle is calculated with the arc tangent function, and depending on the individual values of x and z, the correct quadrant of the projected coordinate is extracted. In `C`, the `atan2` function computes the corrected value of the arctan, which is in the range of $[-\pi,\pi]$. By adding 2π to negative resulting values, a mapping between $[0,2\pi]$ is achieved. As both angles are given in radians, they must be converted to degrees by multiplying them with $\frac{180^\circ}{\pi}$.

The distance value also needs to be transformed to match the values for the sampling scheme, and therefore the distance values are doubled. The main distance from camera and object is based again on the AABB. We use the maximal extend either in x or y direction plus the extend in z direction to correct the distance. Finally, the near clipping plane is added to this distance. This way, the object is positioned correct and the complete AABB is visible after projection. The distance from origin to the camera position is divided by this factor, which results in a distance value usable by a sampling scheme. The distance for sampling is based on a "power of two" progression of the minimal distance value extracted from the object and the projection matrix. To get the corresponding index of this distance value, the logarithm of base 2 is extracted, which can be derived, for example, from the index of the highest bit in an integer representation. After the conversion of the camera coordinates to elevation and azimuth angles, a sampling scheme entry is generated, which is used for lookup of the LCM-Set.

To increase the quality of the extracted conspicuities of the LCM-Set, an interpolation method can be utilized. The most simple and most popular interpolation method for BTF is the nearest-neighbor sampling. This is due to the high sampling rate of the surface information. Other interpolation methods do not sufficiently increase precision while reducing the overall performance of the system. We clamp the current spherical coordinates to the nearest sampling position of the scheme. The corresponding LCM-Set is returned, and is used for saliency calculation. We applied this method to select the LCM-Set in the examples provided in section 'Implementation' on page 130.

As a LCM-Set is provided, the values along the distance can be interpolated as well. It is possible to perform a nearest neighbor selection for the distance values as well as a linear interpolation along the different LCMs. Opposed to interpolations of the complete LCM-Set, here only two single LCMs must be compared and interpolated. As this feature is available directly on the GPU due to the mipmapping of textures, this method will be faster than other interpolation techniques. To enable smooth interpolation based on the distance value, an unclamped floating point distance value linearly weights the according mipmaps of the LCM-Set.

Now that the data of a LCM-Set is available for each vertex during mapping, the conspicuities are extracted by performing a texture lookup within the LCM-Set. After projection, the valid vertex positions $\vec{p}$ will be in Normalized Device Coordinates (NDC), which describe a unit cube in the range

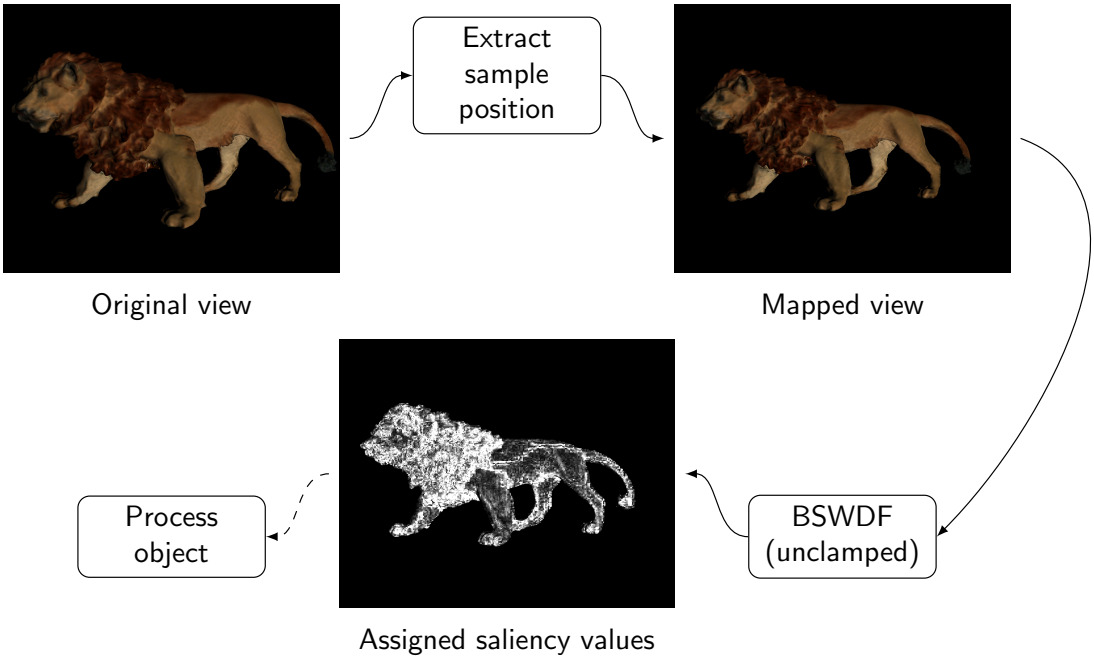


Figure 7.9: The scenario before and after projecting the object into the sample space derived by the sampling scheme. First, the position of the camera and the object are evaluated, and the sampling position is calculated. After the projection, the mapping of the values is performed, and the BSWDF be applied.

of $[-1,1]^3$. By applying a linear offset of $\vec{p} \cdot 0.5 + 0.5$, the position is shifted into the range of $[0,1]^3$. If LCM-mipmapping was applied during the generation of the LCM, these positions have to be adapted to result in correct values. Furthermore, the z-component is omitted and only x- and y-parts are used for the lookup within the LCM to derive the features of the current vertex. The complete scenario of lookup, mapping, and BSWDF calculation is depicted in figure 7.9.

7.6 Evaluation

After definition of the BSWDF, and the presentation of notes regarding an implementation, the proposed methods are evaluated. We will theoretically evaluate the space consumption of a single lookup table. This size is important, and should be small to allow efficient storage on disk. Here, the LCM-mipmap approach will be included as well. Opposed to the storage approach, it may not be reasonable to preprocess the LCM if the representation is not static. For this reason, we also perform time measurements to evaluate our prototype and the presented calculation methods.

7.6.1 Space Requirements of the Lookup

In the following, we assume a LCM with a resolution of 256×256 pixels. When extracting four features, e.g. color differences, luminance differences, curvature, and the silhouette, we can derive

the required memory consumption on disk. Using the 77-sampling scheme with 5 depth samples, a lookup table will consume 385 MiB. This is easily validated by performing the following calculations:

The number of pixels per LCM is $256 \times 256 = 65536$, and each pixel stores 4-floating point values – one for each feature –, which results in 262144 floating point values. Each float consumes 4-bytes, and thus a single LCM requires 1048576-bytes, which is 1 MiB. The complete object is covered with 77 samples at 5 different distances per samples and thus 385 LCMs are generated. In combination with the space consumption of 1 MiB per LCM, the previously stated 385 MiB are derived.

This value is reduced by leveraging mipmap approach for a LCM-Set. Instead of storing each distance LCM with the same resolution, each subsequent version is effectively quartered in size. The required space for a mipmapped LCM-Set is so reduced to $1024 + 256 + 64 + 16 + 4 = 1364$ KiB instead of 5120 KiB for the full resolution. This is approximately 26.6% of the original LCM-Set size, and no important information has been stripped away from the lookup table. The final lookup table is almost quartered in size because storing the complete lookup table with the mipmapped versions requires $1364 \cdot 77 = 105028$ KiB or 102.56 MiB. The used resolution of 256×256 pixels is rather high for saliency images, e.g. Walther and Koch create by default a 64×64 map. A further reduction is possible by generating smaller LCMs. However, the ideal size of a lookup table and the LCMs needs to be determined.

For comparison to these values, a direct storage of saliency values, in dependency of light and camera positions, would require to generate $77 \cdot 77 = 5929$ entries. To create the same detail as with our proposed method, a 256×256 saliency map needs to be generated, resulting in a complete size of the lookup table of $5929 \cdot 256$ KiB = 1517824 KiB (1.4475 GiB) without the depth versions. If they are included to (again five versions), a lookup table requires 7.23 GiB. Without the proposed method, a lookup table approach would not be reasonable at all.

Another way of reducing the required size is to encode the values as a byte representation instead of floating points. This reduces the size by factor 4 while a compression of the data is still possible. As some values do not store any information, i.e. all conspicuities are zero, they can be removed without loss of information as well. All mentioned compression methods of the lookup table, however, remain future work.

7.6.2 Timing

In the second evaluation, the time needed to calculate a single LCM is presented. Despite the fact that the time needed for rendering is not relevant for the solely image-based method, these results are still presented with an additional timing value. During frame generation, the Phong splatting method is used (refer to section ‘Point-based Rendering Methods’ on page 47). All tests have been performed on the test system 1 (see appendix ‘System Configurations’ on page 211).

Two tests were performed, one representing the averaged time required to compute an individual LCM while the second method offers insight into the performed operations of each calculation method. In our implementation, the GPU version applies the normal variation approach to simultane-

Model	Method	Single Generation			
		64x64	128x128	256x256	512x512
Bunny (36 K)	GPU	3.14 ms	4.05 ms	9.55 ms	31.59 ms
	CPU	180.38 ms	644.46 ms	2401.21 ms	9299.65 ms
Lion (183 K)	GPU	10.00 ms	10.38 ms	14.63 ms	35.99 ms
	CPU	172.12 ms	638.96 ms	2352.30 ms	9354.80 ms
Buddha (1 M)	GPU	47.38 ms	46.28 ms	48.84 ms	71.24 ms
	CPU	204.33 ms	667.38 ms	2534.83 ms	9492.03 ms
Dragon (1.3 M)	GPU	57.40 ms	62.05 ms	57.12 ms	75.20 ms
	CPU	203.53 ms	669.04 ms	2385.79 ms	9389.91 ms

Model	Method	Complete Generation with 77-scheme			
		64x64	128x128	256x256	512x512
Bunny (36 K)	GPU	1.3 s	1.5 s	3.6 s	12.1 s
	CPU	69.6 s	4.1 m	15.44 m	59.8 m
Lion (183 K)	GPU	3.8 s	4.0 s	5.6 s	13.8 s
	CPU	66.4 s	4.1 m	15.1 m	$>$ 1h
Buddha (1 M)	GPU	18.2 s	17.8 s	18.8 s	27.5 s
	CPU	78.8 s	4.3 m	16.3 m	$>$ 1h
Dragon (1.3 M)	GPU	22.1 s	23.9 s	22.0 s	29.0 s
	CPU	78.5 s	4.3 m	15.3 m	$>$ 1h

Table 7.3: Timing results of the presented saliency calculation methods. The image-based approach generates the saliency information while the GPU-approach provides the individual conspicuities. Both can be stored within a lookup table and can be used by a BSWDF in any lighting scenario if only object-related information is extracted.

ously estimate the curvature and the silhouette (see sections 'Mapping of 2d Features' and 'Calculation on the GPU').

The timing results, shown in table 7.3, indicate that both approaches can generate saliency information at almost interactive frame rates for small input sizes – 30 FPS are approximately 33ms of calculation time. In the listed timing values, the complete generation of the saliency information is reported including a draw of an object. However, if the features are calculated during normal rendering, this additional drawing may be omitted. As the GPU method scales better than the image-based approach, a further optimization could result in a full interactive solution. Both methods also allow to store the calculated features in a lookup table as long as object-based information is computed (refer to section 'Creating a Lookup').

In table 7.4, the relative processing times for generating the final values are shown. In the image-based method, only a small fraction is spent for generating the input image while most time is needed to calculate the orientation features. This is due to to the applied Gabor filter. In the GPU-based approach, the expensive approximation of these orientation features is circumvented, and most time is required for normalizing the values gained from the feature extraction and DOG application. In our prototype, the normalization step is made by the CPU using SSE instructions. Iterative normalization

Size	Input	Features	SaliencyMap
64 x 64	9.45%	90.17%	0.21%
128 x 128	8.01%	91.67%	0.13%
256 x 256	3.93%	95.77%	0.11%
512 x 512	2.39%	97.31%	0.11%

(a) Distribution of processing for the image-based method

Size	Splatting	Mipmaps	Features	Normalize
64 x 64	77.46%	1.67%	1.13%	19.53%
128 x 128	71.88%	2.31%	1.07%	24.43%
256 x 256	53.94%	3.87%	0.83%	40.30%
512 x 512	31.02%	6.01%	0.36%	60.62%

(b) Distribution of processing for the GPU-based method

Table 7.4: Distribution of the individual calls performed by each extractor. The values denote the fraction of the complete saliency or feature map derivation required in the lookup table generation. For testing purposes, a complete lookup table was generated using the 77-sampling scheme. The shown results are averaged over the entire lookup table generation. The Stanford lion was used in all cases.

methods could also be transferred onto the GPU, accelerating the overall process.

7.7 Conclusion

The BSWDF allows to calculate a saliency value for a visible surfel of an object in question. By using these saliency values as priorities, e.g. by mapping these onto surfels or primitives, a selection based on perceptual information can be achieved. Due to the universal design of the BSWDF, a calculation for both 2d images and 3d objects is possible.

Saliency covers the early vision of humans that begins at the retina and ends at the LGN and V1. There a selection of the most important signals is done. As this information cannot be influenced by a spectator, it allows application without any a priori knowledge of the scene and the performed task.

The well established 2d saliency model has often been proven valid, and several enhancements towards better performance and quality of the results have been made. But until now, a complete 3d saliency model had not been defined. With the BSWDF, we introduce a 3d mapping of the operators and present a complete 3d visual salience model. The mapping of the operators emulates and extracts the same features as in the original 2d model. Additionally, the individual feature extractors take advantage of the 3d space and its corresponding representation. This allows to directly calculate or extract features by using simple mathematical instructions. For example, in case of the motion of an primitive or object, only the displacement in 3d needs to be calculated. We presented extraction methods that have been implemented for CPU and GPU as well. The speedup achieved by the GPU method is significant, but might be enhanced even further by developing highly specialized shaders and optimizing the rendering.

A lookup table approach has been presented because large computational effort is needed to create a conspicuity map. Our extraction methods allow to generate a LCM in advance, saving computation time. This also avoids recalculation of features that already have been extracted as the lookup table can be constructed during run-time. Furthermore, an efficient storage is enabled because only object-specific information is stored and illumination information is added during BSWDF computation.

Initially, the idea of using the Phong splatting algorithm occurs to be not optimal as multiple rendering passes are required. However, the accumulation of individual splats directly provides the required scale information for the feature calculation. Other primitive types can be utilized as well because the extraction relies only on surface properties. The process of accumulation is independent of the representation.

The visual salience values that are derived from the LCM with the BSWDF should provide valuable information regarding perception, and thus the proposed BSWDF was tested in a user study. The results indicate that the system has the ability to identify important regions. More details are presented in section 'Evaluation' on page 135.

The BSWDF is the foundation for the following systems to be presented, and this enables adaptation of 3d objects. By omitting or adding individual feature extractors and weighting these features individually, a free combination and situation-based enhancement of single features is possible. Furthermore, some features, i.e. surface related information, can be calculated in advance, which reduces evaluation costs. To compute the final saliency values, only few instructions are required, and the BSWDF can be implemented in a shader-program.

Yet, the approach can be improved, and several enhancements are proposed. The BSWDF has only been tested for a point-based rendering of single objects, which including animation and static images. The applicability of the system should be verified in a complete scene when saliency information needs to be combined. Motion pictures also require temporal evaluation of the features, and thus adaptations to the feature extractors may be needed.

The lookup table does not account for a changing object representation. If the object is morphed, the precalculated data is invalidated. The question arises if this data has to be completely removed. Possibly, some information could be reused without introducing errors. Therefore, a caching strategy could be developed to account for changing representations, and an in-place storage of results could be achieved when calculating preattentive features.

Within the BSWDF, an illumination model needs to correctly scale the preattentive features. For an uncolored light source, this is an easy task, but colored light sources might enhance or reduce the influence of color differences. Therefore, a representation of a light source is required that weights the available, precalculated color differences. This should be possible because the surface features are object bound, and the light source only changes the emitted color. We assume that a difference representation for each illumination model can be derived, which correctly scales these precalculated features.

8. Feedback System

The application of the BSWDF allows to identify important regions on an object's surface within a rendered scene. At this point, the different presented data structures and methods are merged to define a *Feedback System*: a fully dynamic, saliency-influenced rendering system. An object within this *Feedback System* will become self-optimizing with respect to imposed restrictions.

The main operation is an extraction of perceptual information present in the current scene's representation. In this scenario, we will restrict the scene to a single object, which is displayed and altered using the *TreeCut*. A BSWDF is applied to each surfel, and the computed saliency values are interpreted as priorities. These priorities control the *TreeCut* evaluation strategy that adapts an object. When this object is changed, a new view is generated, and thus new saliency information is available. This, however, results in a change of an object's representation, and a feedback is generated.

This feedback changes the representation using saliency information. With the help of the mapping techniques, which were proposed in chapter 'Bidirectional Saliency Weight Distribution Function' on page 99, we are able to apply the BSWDF using the GPU. However, the results need to be transferred to the evaluation strategy of the *TreeCut*. It would be possible to calculate and assign the saliency data on the CPU, but the 3d processing power and pipelining capabilities, which includes parallelization on the GPU, will outperform any CPU approach. In recent years, popularity has increased to process data on the GPU, i.e. GPGPU scenarios. In those cases, an exchange between the CPU and GPU is important, and fast extraction mechanisms have been developed. These so called feedback methods will be applied to acquire the computed information from the GPU.

It was shown in the previous chapter that the generation of saliency features can be performed either on-the-fly or in a preprocess where the results are stored in a lookup table. In the latter case, these need to be transferred to the graphics card to enable BSWDF calculation. Furthermore, the feedback and the *TreeCut* representation need to be synchronized to use the correct data for computation. If both is assured, a reduction performed by the *TreeCut* will decrease the size of an object's representation while in preserves visual details that are important.

The *Feedback System* is a combination of the priority, i.e. saliency, extraction and the parallelized evaluation strategies provided by the *TreeCut*. We utilize the GPU to assign the priorities of the drawn surfels to the current *TreeCut* nodes. A special focus is laid on the extraction of the assigned saliency values. After extraction, the cut-nodes are processed using these priorities. Due to the multi-threaded approach, evaluation can be performed over several frames without stalling rendering. Once the representation has been updated, the new priorities are derived, and the loop is closed. For control of this self-optimizing system, a threshold is utilized to assure the stability of the *Feedback System*. This stability is necessary when no more changes to the representation are expected. This threshold can be altered during run-time, and it controls the influence of the evaluation strategies.

In the next section, the requirements of our *Feedback System* are stated, and our mechanisms for control are introduced. The loop and the emerging self-optimizing system is covered in section 'Feedback Loop'. In section 'Implementation', we will give some notes regarding our created prototype and present the applied GPU feedback method. Finally, the overall system is discussed and its performance is measured. Additionally, user tests are performed to validate the visual quality of the achieved results. This chapter closes with a discussion of suitable enhancements with respect to the presented system.

8.1 Requirements

The *Feedback System* depends on both *TreeCut* and the BSWDF, so some limitations are induced and have to be solved in advance. These limitations concern the rendering as well as our MVC-based framework design. When optimizing a representation, some kind of restriction is required to which the display is adapted to. For example, the *Feedback System* can be restricted to act in real-time, which limits the total time for drawing an object to less-or-equal than 33 ms – in case of 30 FPS. Therefore, the detail of the representation is reduced until the required frames rate is achieved. Alternatively, a maximal primitive count can be used as well.

The problems we encounter when optimizing the representation can be categorized into two important aspects. The first are hardware restrictions implied by the system configuration. However, this configuration is not known a priori, and the system has to dynamically adapt to those restrictions. The second aspect is the self-optimizing system, which needs to converge to a stable representation. As a perceptual-optimized representation depends not only on the object, but is also influenced by the view of it, a once stable and optimal configuration may not be optimal anymore. Thus, a re-optimization is required if the view of the object has been changed.

8.1.1 Limitation by Hardware Resources

Hardware issues, which arise during design and creation of a real-time application, are almost impossible to circumvent. However, when reducing basic operations to a minimum and unnecessary computations are avoided, the processing time is decreased independent of the available hardware. In our case, this includes to fetch data from memory as well as to transfer this data onto a device with limited capacity, e.g. the graphics card memory. While it is preferred to have all the data available on the graphics card, this is not possible in typical environments as the huge amount of data easily exceeds the available memory.

Thus, the lookup of the LCMs, required by the BSWDF for evaluation, is only performed when necessary. This avoids costly uploads and memory accesses. The scene information is utilized to determine whether an upload is required, e.g. if another lookup table entry is required. However, this upload is unnecessary if the stored features need to be recalculated. This is the case if either the data has not been precalculated or it is considered invalid, for example, because a simulation is applied to

an object. As this re-computation is quite expensive, it should only be performed when new data is required by an evaluation algorithm.

When taking advantage of the graphics card to accelerate the rendering and computation of priorities, the gathered information needs to be extracted. This transfer is expensive and stalls the rendering. This extraction is only required by an evaluation strategy, so it will only be performed when an iteration of the evaluation has been completed The information that is acquired during a running evaluation cannot be extracted since the access to internal data of an evaluator is locked. Therefore, an extraction is triggered after exchanging the new representation. This also assures that the current surface is used for priority assignment.

Summing up the hardware limitations, the biggest bottleneck is to copy the necessary data. The CPU has to transfer saliency information of the current view to the graphics card in form of LCMs. The GPU returns the calculated saliency values back to the CPU where it is processed by an evaluation strategy of the *TreeCut*. A possible approach is to reduce the impact of this transfer by using smaller data types. For example, it would be possible to transfer 16-bit or 8-bit information from the graphics card to the CPU. With floating point values, the calculated priorities of an object with 64.000 nodes would require to transfer 256 KiB each time the extraction is forced. Using only 8-Bit information, this amount is quartered yielding 32 KiB. The loss in precision would not invalidate the results because our priority-based evaluation strategy only applies a partial sorting. Also, the required information could be stored with an compression method while still reducing the transfer costs. These additional methods admit to improvement of the presented approach, but are beyond the focus of this work.

8.1.2 Controlling the Loop

Due to the self-optimizing property of our system, the representation of the object is optimized with respect to the gathered perceptual information. However, the saliency values derived by the BSWDF do not only depend on the features extracted, but also on the current view. This means that a change of the view requires the current representation to be adapted accordingly. It is also necessary that the system converges to a stable configuration, i.e. no changes will be applied by the *TreeCut* after all important changes have been made. We propose a threshold that controls the minimal gain in quality of the representation. If a *TreeCut*-operation will not increase the detail up to this threshold, it will not be performed. For example, this is the case when a `coarse`-operation's penalty plus the threshold is larger than the pending `refine`-operation's gain. An iteration is safely aborted when a cut-node has been rejected this way because nodes are processed with descending priorities.

When the scene or view of an object is changed, the according LCMs need to be available. As stated before, an upload is only necessary when new LCMs are required. We therefore use the properties of our lookup table sampling schemes (refer to section 'Sampling Schemes' on page 114). The current WVP matrix of an object is evaluated to derive the according entry of the lookup table. If this entry is different than the current entry, new data will uploaded to the graphics card. This check needs to be performed every time when the evaluation strategy has completed processing. Furthermore, new

priorities will only be evaluated by the GPU when data, e.g. the index-list, has been exchanged. The feedback is only activated when either process has requested new data.

8.2 Feedback Loop

The feedback loop defines how the individual parts, e.g. the *TreeCut* and the BSWDF, affect each other. This allows one to control those and to achieve a specific behavior within a global environment. Here, perceptual information is used to determine priority values, and thus the evaluation strategies account for visual quality when applying the *TreeCut*-operations. In the following, some details regarding the *Feedback System* are presented.

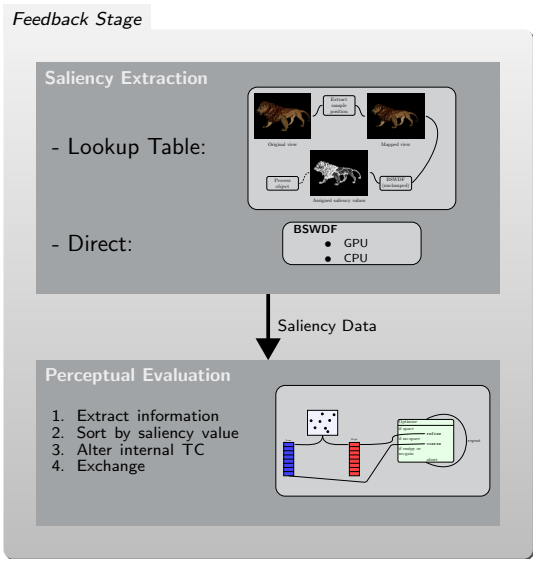


Figure 8.1: Outline of the proposed *Feedback System*. The feedback starts with an extraction of the saliency for the current representation. This is either based on a lookup table or a direct evaluation using the last frame. This information can be retrieved from the renderer. The evaluation algorithm calculates the new representation and may be exchanged. During the feedback, regular drawing is continued with the old representation.

As the BSWDF is being computed during the rendering of the object, the feedback information is actually one frame behind. The evaluated data does not reflect the current scene, but this can be considered harmless in common scenarios. Multiple frames will be drawn before a new representation of an object is available because the evaluation algorithms are parallelized. The differences between consecutive frames are not too large – assuming a real-time application –, and the calculation results remain valid. In rendering scenarios, this is referred to as frame coherency, which is given if the data between two frames is almost identical. Usually, only small changes are applied between individual frames, e.g. by rotating or moving the object slightly, which generates frame coherency.

The *Feedback System* manages the exchange of the required data between the individual threads. After extraction of the priority values, the evaluation thread is invoked with the new data, and a new representation is generated. When the evaluation has completed, the index-list maintained by the evaluation strategy is copied to the *TreeCut*. This results in an updated surface representation because the object is displayed using this index-list (refer to section ‘Multi-Threading’ on page 85 and section ‘Data Separation’ on page 80). After the exchange, the feedback mechanisms for extraction of priority values are activated (refer to section ’Using GPU for Data Feedback’). If necessary, new LCMs are uploaded as discussed before, which are identified by the lookup table entry derived from the current object’s view. Following this

upload, the BSWDF is applied, and the new priorities are derived.

In figure 8.1, a simplified version of the loop created by the *TreeCut*- and the BSWDF-evaluation is depicted. The *Model* provides the current representation. The BSWDF calculates the according priority values, which are used by the *TreeCut*-evaluation to modify the current representation. The loop closes with an exchange of the new information. This minimalistic loop can be extended by introducing external effects, such as the animation of an object. In this case, not only the view but the animation itself can force a re-optimization of the representation. The visual evaluation and the adaptation of the object build the two main parts of our dynamic *Feedback System*.

8.2.1 Self-Optimizing System

By encapsulating the feedback loop, the *Feedback System* can be understood as a self-optimizing system. The *TreeCut* optimizes its representation using the available saliency values. This adaptation, of course, only occurs if a limitation or restriction is imposed.

A self-optimization system, however, has one major drawback: A stable configuration is not easy to achieve, and this has to be accounted for during design. After `refine`-ing a node and computing new saliency values, it is possible that these newly introduced nodes are considered unimportant and are `coarsen`ed afterwards. This introduces a flicker in the representation because surfels appear and disappear during display. Even if a smooth transition between the nodes is applied, which reduces the impact of the flicker, this behavior is still undesirable.

As stated in 'Controlling the Loop', we propose the use of a threshold value to assure a static representation of an object, i.e. to ignore possible changes that do not introduce enough gain. We therefore extend the *TreeCut* evaluation strategies with the threshold value and compare the gain of each *TreeCut*-operation against it. Once an iteration has been aborted due to this threshold, the evaluation will not be performed until new data is available. It is possible to modify the threshold during run-time, e.g. to increase the influence of lower priorities. However, this could lead to the unwanted flicker again.

As stated before, the self-optimization allows the complete system to adapt to a global environment, but this comes at the cost of having to control a highly dynamic system. Thus, the system will never be able to compete against a manually tuned implementations that use fixed representations. Yet, the *Feedback System* is able to adapt to different scenarios. This adaptation is done with respect to the available hardware resources. The overhead introduced by the dynamic system depends on the performance of the evaluation methods used for both the *TreeCut* and the BSWDF. However, the parallel implementation assures interactivity as long as the hardware is capable of drawing the current object representation at interactive frame rates.

8.2.2 Priority Extraction and Feature Map Upload

In the following, the computation and extraction of priorities as well as the supply of the required LCMs to the graphics card will be referred to as the Dynamic Feedback Method (DFM). To reduce

the transfer of data to a minimum, this will only be enabled if a new surface representation is available. When initializing the *Feedback System*, the DFM has to be activated as well.

To allow the graphics card to compute the BSWDF and the corresponding saliency values of the current representation, the according LCMs must be accessible by the GPU. As stated before, a WVP matrix is extracted with the help of the sampling scheme that was applied during lookup table generation. In our prototype, a nearest neighbor method is used to construct a LCM-Set. We utilize the capabilities of graphics cards to interpolate between single maps of the LCM-Set.

The saliency values are extracted with the current view as input. The view is projected onto the LCM and matched with the entries within the data (refer to section 'Lookup The Data' on page 117). To extract the computed priorities, the *Feedback System* has to provide a buffer for writing in a graphics card compatible format. The buffer declarations and definitions are specific for each rendering subsystem. In our case of openGL, either a "texture buffer object" or a "transform feedback" method can be utilized to transfer the large amount of data from the GPU.

8.3 Implementation

Based on the definition of the *Feedback System*, a prototype of the system has been implemented in `C++`. We will give some technical notes regarding the implementation of this prototype, and we explain how to circumvent possible technical difficulties. Our prototype displays and optimizes a single object, but the complete framework is not limited to a certain number of perception-based objects.

The BSWDF allows to calculate the saliency values for the current scene. By applying the splatting approach, the necessary information is accumulated and the conspicuities are extracted from the GPU. After the normalization, a LCM is transferred to the GPU again to feedback the final priority values for each splat. These are forwarded to the evaluation strategies of the *TreeCut* and the scene is adapted. We refer to this method as the Direct Saliency Calculation (DSC).

The lookup table, as proposed for the BSWDF (refer section 'Creating a Lookup'), accelerates this process. It is a data provider for the computation of the BSWDF and can access the precomputed LCMs. As stated earlier, there is no need for the lookup table to be initialized in advance as it is writable during run-time. This, of course, comes with some performance penalty because the data has to be computed before it can be accessed by the BSWDF. This method will be referred to as the Lookup Table Saliency Calculation (LUTSC).

A *TreeCut* is generated upon a multi-resolution object, and for evaluation, a priority-based strategy is selected. To restrict the display in the complete system, a maximal number of surfels can be defined. The *Feedback System* reduces the detail of the shown object until the number of its primitives is below that number. In our prototype, multiple types of priorities can be selected, and we provide three basic methods to reduce the detail: "Saliency" (SAL), "Point-Size" (PS) and "Saliency Enhanced Point-Size" (SALPS) reduction. The plain reduction uses only the values extracted from a BSWDF

```
1  //First extract camera position
2  Vector4 camPos = Vector4(GetCameraPostition());
3  //Correct camera position using offsets of lut
4  Vector3 offset = lut->GetModelOffset();
5  camPos.GetX() -= offset.GetX();
6  camPos.GetY() -= offset.GetY();
7  //Get object matrix and invert
8  Matrix4x4 objectMatrix = GetObjectMatrix();
9  objectMatrix.Inverse();
10 //Generate lookup from camera
11 if(lut->GetSamplingScheme()->SetFromCameraPos(camPos))
12     uploadLCMSet();
```

Listing 8.1: Method for computation of a lookup table entry using the current camera position and object matrix. We extract camera and object information to derive the camera position in object space.

evaluation. The PS reduction uses solely geometric information in form of surfel's point-size. The last method, SALPS, is a combination of both methods.

8.3.1 Deriving the Lookup Table Entry

An object is rendered and displayed using customized World, View, and Projection matrices, so the projected position of the object cannot simply be assigned to a lookup table entry. This is due to the fact that the stored information has been transformed based on the sampling scheme that was used during lookup table creation (see section 'Sampling Schemes' on page 114). In addition, there is no guarantee that the projection matrix matches the projection used during generation. Both must be corrected before an lookup table entry can be derived.

The 3d data needs be transformed with the same information that were used during the generation of the lookup table data. We first recapitulate the generation of a lookup table entry: The lookup entries describe a view around an object on a unit sphere. The entries are identified by elevation and azimuth angles as well as a distance value. Thus, we need to convert the current World and View matrices to a sphere representation. Then, the corrected coordinates, i.e. azimuth and elevation, can be calculated to derive the according lookup table entry.

In each lookup table, a global projection matrix along with local object offsets are stored. We invert the object's specific transformation matrix, i.e. the World matrix, to project the camera's current position into the object space. We use the capabilities of the sampling schemes to create and update the current entry with the new camera position (refer to section 'Lookup The Data' on page 117 for a detailed description). If a new lookup table entry is derived, the according LCM-Set is uploaded. In listing 8.1, a pseudocode is shown that performs the required computations.

8.3.2 Using GPU for Data Feedback

For extraction of the LCM values, we leverage the capabilities of shader-programs. These allow to process custom instructions on the GPU.

If the DSC is used, the method explained in 'Calculation on the GPU' on page 110 is applied. The current scene provides the necessary information required to derive a LCM. In case of the DSC, two projections will be performed: One for generating the visual output and the second for mapping of the LCMs to the current representations. This way, geometry data needs to be processed only once, and in pre-tests, no significant difference has been measured with respect to the processing times. Thus, we have fulfilled a requirement of the lookup table usage (refer to 'Lookup The Data' on page 117).

Lookup Table Processing

The LUTSC-approach requires some additional computations. In contrast to the DSC, the stored information must be transformed before the information can be used.

The lookup in the LCM requires a projection back into the coordinate system used during generation. This is done by the sampling scheme provided. A WVP matrix is extracted for the view. The projection matrix is stored in the lookup table along with other information required. The WVP matrix is passed to the shader-program, and the input primitives are transformed accordingly.

To account for LCM-mipmapping in a lookup table, the distance value is stored in a global shader-parameter. The unclamped index value of the sampling scheme allows a smooth interpolation and increases the accuracy of the method. While different types can be used, we apply a linear interpolation of the values in our prototype.

The projection of the primitive's vertices transforms the coordinates into NDC. Backfacing vertices are identified by projecting the surface normal and by comparing the resulting z-values. We apply backface-culling and assign a priority of 0 to these vertices because the values that are stored in the LCMs do not belong to them. Is a vertex visible, however, the NDC are transformed into LCM space. This is achieved by defining a viewport mapping based on the LCM extends. If the lookup table entries contain LCM-mipmaps, the coordinates are adapted accordingly by applying a scale of 2^{depth} with depth being the index of the LCM-mipmap.

Calculation of Priorities

At this stage, both methods provide a one-to-one mapping of the calculated conspicuities to the visible vertices of the representation.

The evaluation of the BSWDF on the GPU is then straightforward. Surface normals and light information are calculated using common methods in World-View- or World-space coordinates. The texture fetch of a LCM is performed with the derived texels, and the weighting of the individual conspicuities is implemented by computing a dot-product.

Mipmapping of the basic 2d textures, which is only necessary for the lookup table approach, is

supported by the graphics hardware. Both the selection and the interpolation is done internally. A non-mipmapped version requires either a 3d texture or an array of 2d textures. The former is common in rendering subsystems while the latter is available in openGL as an extension.

8.3.3 Transfer from GPU to CPU

After application of the BSWDF, the computed priorities need to be transferred back. In openGL two methods are available to extract calculated data from the graphics card to the main memory. The first approach is to use the so called *texture buffer object*s, which allow to read and write into textures and operate like normal buffers. After loading the values into a *texture buffer object*, they can be accessed by a evaluation strategy. A drawback of this method is that the individual positions within the buffer must be defined appropriately to allow an identification of the vertices. This can be achieved using the VERTEXID property of a vertex.

Another, more handy method is the *transform feedback*, which was mainly developed for the growing GPGPU community. Vertex data, which is processed either from a vertex- or a geometry shader, can be written into a buffer created by openGL. This is achieved by assigning a value to a predefined vertex property within a shader. These properties are extracted by the rendering subsystem and are mapped into a buffer. This approach has the advantage that the primitives are ordered by the VERTEXID property by default. No manual tweaking of the buffer positions is necessary.

Both buffer types require to know the size of the them in advance. Otherwise, the computed data will not be stored correctly and will be discarded silently. As the size of the *TreeCut* is fixed during rendering, the buffers can be created or adapted accordingly. These do not need to match the size of the *TreeCut*, and additional memory can be allocated if necessary. Using the *TreeCut*'s maximal size allows to avoid unnecessary calls to the openGL API, and this bypasses expensive allocations during rendering. Both methods are mapped into the main memory as an array or a *pixel buffer object*. As the *transform feedback* method inherently writes the generated primitives in the order they were inserted into the pipeline, no manual sorting has to be performed. Another advantage of the *transform feedback* method is that the number of generated and written primitives is obtainable with openGL queries. These queries are useful for debugging purposes and allow to check whether the buffer was filled correctly or data has been cropped due to the lack of capacity in the buffer. If the written primitives count differs from the number of generated primitive, an overrun occurred, and not all results were stored in the buffer. In this case, a resize of the buffer is required, and another computation by the BSWDF needs to be performed. As opposed to the *transform feedback*, such queries are not available for the *texture buffer object*. If not enough space for writing results is available, these are silently discarded.

To apply the *transform feedback*, a set of attributes has to be defined, which will be written into the allocated buffer during the vertex or geometry shader stage. The attributes can be extracted in an interleaved or a sequential manner. The sequential order creates an attribute-oriented layout while the interleaved order is vertex-based. The number of components that are extracted for each attribute

needs to be defined as well. This number is normally in the range of [1,4]. For example, one can choose to extract four values from the position attribute to provide these as an input for the next pass. So, several iterations are possible to generate a fractal object. Almost all attributes are selectable for transfer, and the complete list of the attributes is defined by the *transform feedback* extension of openGL.

As a last step, the feedback needs to be enabled via an API call. Both the buffer and the attributes, which will be transferred, can change dynamically, but this is not necessary in our scenario. The draw mode and the primitive type has to be selected as a parameter. This assures correct extraction of the attributes because they vary with the used primitive type.

After the rendering pass, the feedback is disabled, and the following rendering passes are executed without the DFM. The buffer is mapped into main memory, and access by the evaluator of the *TreeCut* is enabled. Before the next transfer of attributes, the buffer must be unmapped to avoid errors.

We include a special vertex attribute, which stores the calculated BSWDF results. These are one-valued floats and have the same number as the used vertices for rendering. The DFM thus scales directly with the complexity of the displayed scene.

8.3.4 *TreeCut* and Feedback Synchronization

As stated in section ‘Multi-Threading’ on page 85, the evaluation strategy of the *TreeCut* is executed in parallel to the rendering process. But, it has to be assured that the saliency value buffer is available for reading during evaluation. This data, however, is only calculated if the DFM has been enabled.

While the DFM is active, the evaluation remains in `wait`-stage. The rendering is performed using the *TreeCut*'s current surfels. Afterwards, the data is mapped from the graphics card memory to the main memory. With this mapping, the evaluator process can access the saliency values, and thus the evaluator thread is invoked. As the current cut has been used for rendering, the acquired saliency value buffer is aligned with the surfels. This enables a fast extraction of the surfels' priorities.

With these enhancements to the proposed multi-threading approach of the *TreeCut*, rendering remains independent from the evaluation. Within the rendering thread, a query is used to determine whether the evaluation has completed and a new index-list is available. When this query indicates the completion of an iteration, the index-list is exchanged and the DFM is activated again.

If the LUTSC-approach is used, the lookup table entry is recalculated, a new LCM-Set is uploaded. Also the projection matrix required for mapping is calculated and set in the according vertex shader. The DSC-method does not require such updates. Instead, the current scene is processed by the renderer and the conspicuities are combined on the fly.

After derivation of the LCM or LCM-Set, the DFM is enabled. In the next rendering step, the BSWDF is applied, which calculates the new saliency values of the current representation. Once these are mapped into a local buffer, the evaluation is invoked. As long as this is generating a new representation, the old is displayed.

Speedups are achieved when reducing the number of exchanges regarding the index-list, i.e. a new

representation. The threshold approach allows to modify the impact of the saliency values within the evaluation. The larger the threshold, the less primitives are exchanged. Thus, the overall method is accelerated. However, accuracy and detail needs to be trade-off against processing speed. The size of the LCMs and LCM-Sets is also important as they must be either transferred or generated during recalculation.

8.4 Evaluation

Several tests regarding the *Feedback System*'s performance and the resulting preservation of visual quality are presented. We show that the *Feedback System* is capable of rendering at interactive frame rates despite the additional transfer due to the DFM. We therefore compare the *Feedback System* to a plain rendering approach. We also show that the visual quality during adaptation is retained when using the saliency-influenced reduction methods.

In the first test, we focused on performance penalties or overhead when using the proposed *Feedback System*. These penalties arise due to several factors within the DFM, which include the *transform feedback*, the LCM-Set upload, or the generation of the according LCM in case of the DSC-method. We compare the number of drawn frames and number of frames with the active DFM to show that the increase is not required every frame. Still the methods are fast enough to remain interactive. These results are in dependency to the size of the *TreeCut* and will support the assumption that the overhead is generated only rarely. Thus the rendering is not penalized.

We conducted different user tests to rate the visual quality of our approach. Therefore, a single object, which was modified using our *Feedback System*, was shown to a user and was compared to a another LOD-version of the object. Different reduction strategies were applied ranging from plain geometric to saliency-influenced reductions. The user tests confirm the improvements achieved when applying the *Feedback System* in combination with a saliency-enhanced reduction.

8.4.1 Timing Results

The dynamic *Feedback System* is designed to perform an online improvement of an object. Several time measurements were performed to show the real-time update capabilities of the proposed system.

In each test, a tree-based representation of the objects was used. Its generation, however, is excluded from the tests. In case of the lookup table, a LCM-Set consists of mipmapped LCMs with size 256x256 at level 0, while in the direct method only one LCM is generated with a size of 256x256. During the test, only one object is displayed. The object is rotated, translated and reduced in size, while the light is spinning around it. For all tests, we used the four default objects available at the Stanford repository, i.e. lion, dragon, bunny and buddha. The impact of the DFM, which, for example, includes lookup table calculations, e.g. WVP matrix derivation, is compared to a plain rendering, i.e. when the *TreeCut*-evaluation is running or finished and no exchange of data is necessary.

In table 8.1, the percentage for each draw method with active DFM is shown in comparison to

Method	# (Ex)	Time (Ex)	#	Time	Calls	Increase
Test System 1						
DSC	727	11.72ms	5033	3.11ms	14.44%	376.14%
DSC (Phong)	1476	16.38ms	4284	9.50ms	34.45%	172.31%
LUTSC	212	4.91ms	5548	2.70ms	3.82%	181.70%
LUTSC (Phong)	195	10.59ms	5565	8.33ms	3.50%	127.08%
Test System 2						
DSC	296	16.04ms	5468	2.59ms	5.41%	618.63%
DSC (Phong)	627	17.86ms	5137	4.63ms	12.21%	385.84%
LUTSC	153	6.27ms	5611	2.48ms	2.73%	252.08%
LUTSC (Phong)	205	9.31ms	5559	4.43ms	3.69%	210.20%

Table 8.1: The measured overhead of the DFM in the *Feedback System*. Both Increase and number of calls are given in percentage. # (Ex) and Time (Ex) denote the measurements taken during an forced activation of the DFM, while # and Time are taken in the normal, deactivated case. The #'s show the count of the individual calls, while the Times are averaged over the complete simulation. The DSC-method computes the saliency values, while LUTSC is based upon a lookup table. In the latter case, the WVP recalculation and upload of the LCM-Set is included as well. When using the Phong splatting method, more calls are made using DFM because the performance of the evaluator is independent of the rendering.

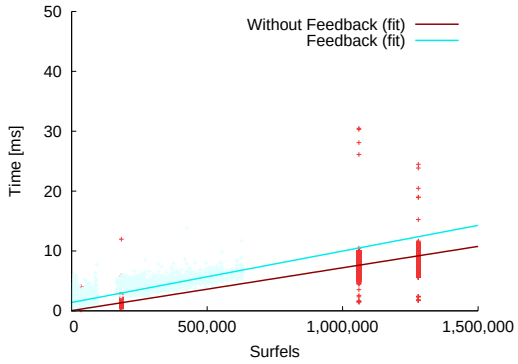
the deactivated execution. The time values are averaged over multiple iterations and were taken on the test systems 1 and 2 (see appendix 'System Configurations' on page 211). The results show that the rendering is not penalized too hard when activating the DFM. Obviously, the increase in with the direct method is extremely large, but this method also generates the most exact saliency values. The results were made with a partial sorting size of 8192 elements. The larger number of exchanges in the direct methods is due to the longer computation in the main thread, which allows the parallel evaluation to compute more versions.

We visualized the increase of the used method of our DFM. In figure 8.2, the rendering times, which are dependent of the size of the *TreeCut*, and the increase with activated DFM is shown. In the first graph, the Plain splatting is shown whereas the second contains the Phong splatting rendering timings.

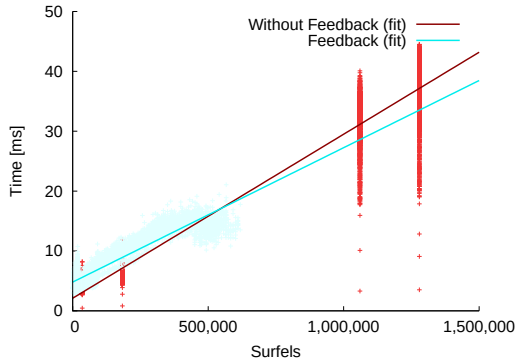
Finally, we compared the DSC-method with the LUTSC-approach. The results in table 8.1 indicate that the LUTSC-method provides a higher performance. In the graphs shown in figure 8.3 visualize the higher performance of the LUTSC-approach.

All results indicate that the system is capable of manipulating an object online while retaining interactive frame rates. The time until a stable configuration is reached depends directly on the performance of used *TreeCut*-evaluator strategy. In table 8.2, we list the distribution among the individual operations that are performed by both *TreeCut*-based evaluation strategies.

The priority-based evaluation spends, on average, more than 50% of the complete calculation time for sorting the input primitives. During this sorting by highest and lowest priorities, an insertion



(a) Comparison between plain rendering with and without feedback enabled.



(b) Comparison between Phong rendering with and without feedback enabled.

Figure 8.2: Comparison between enabled and disabled DFM and the operations in the *Feedback System*. The figures show the individual rendering times for Plain and Phong splatting over the complete simulation. The difference between both saliency calculation methods is too small and therefore omitted in this figure. The results were taken on test system 1.

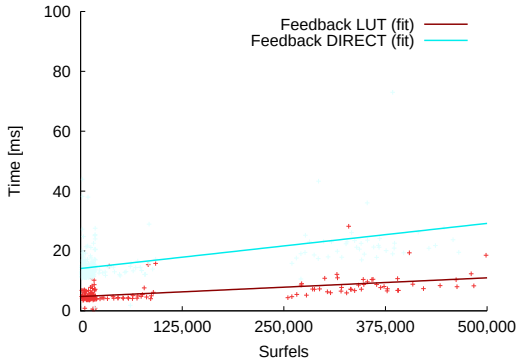
into the partial-sorted list is required[1]. The more primitives are to be sorted, the more time will be spent for insertion. In our prototype, we found a partial-sorting size of 8192 or 16384 primitives to be ideal. In this case, approximately 50% of the time is spent for sorting. The extraction of the necessary information greatly benefits from modern GPUs. Despite the slower CPU on test system 2 (see 'System Configurations' on page 211), the transfer of the data is performed faster.

In case of the bucket-based evaluation, the data throughput is higher as the maximal count of altered cut-nodes is not limited. This approach is also not restricted by the LCM-Sets and the accompanied transfers. We measured that almost 50% of the time is required for determining the bucket of

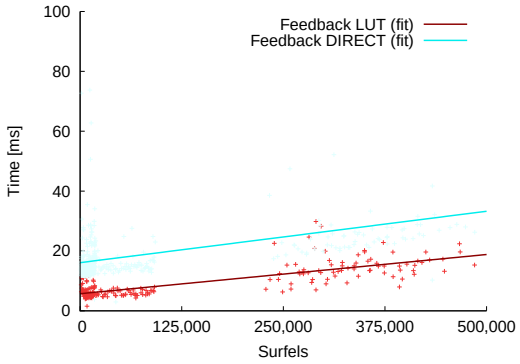
[1] In our prototype, we are using a heap-sort to create the sorted list.

Priority-based Evaluation			
Partial-Sorting Size	Extract	Sort	TC-Operations
1024	15.80%/7.30%	55.91%/70.50%	1.01%/1.81%
2048	15.83%/7.36%	55.86%/68.76%	1.36%/2.28%
4096	15.68%/6.98%	53.46%/65.63%	1.76%/2.73%
8192	14.76%/7.77%	50.06%/60.49%	2.14%/2.93%
16384	15.05%/8.02%	49.91%/58.32%	2.49%/3.33%
32768	16.47%/7.45%	48.76%/60.01%	2.63%/3.62%
Bucket-based Evaluation			
Bucket Selection		TC-Operations	
53.72%/57.66%		9.89%/12.34%	

Table 8.2: Impact of the operations performed in each evaluator. The Extraction shows the fraction of time spent for acquiring the data from the rendering. The sorting that is performed requires most of the time in the priority-based evaluation. The Bucket Selection is the bottleneck in the bucket-based evaluation. The left values are taken from test system 1, while the right ones are from test system 2.



(a) Plain splatting performance.



(b) Phong splatting performance

Figure 8.3: Comparison of the DSC- and LUTSC-based DFM. The Direct-methods has a higher computational demand than the lookup table approach. However, the saliency values do not need to be mapped, which is necessary in the lookup table approach. These results were taken on test system 2.

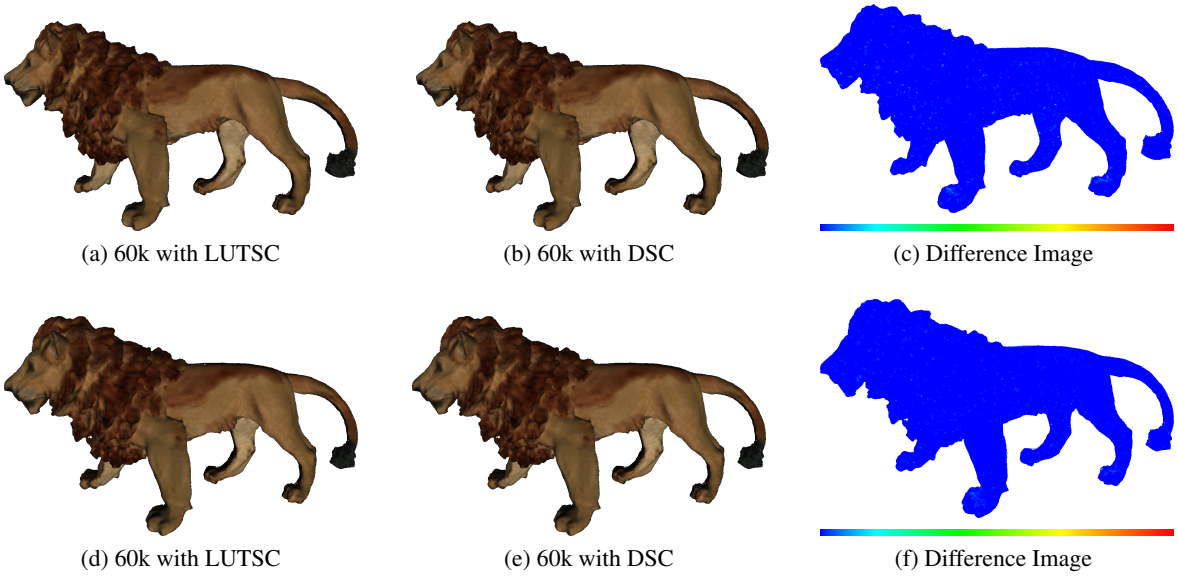
(a) 1060k (b) 500k (c) 250k (d) 100k (e) 50k (f) 25k (g) 10k

Figure 8.4: A LOD-progression of the Stanford buddha. The selection of surfels is performed online based on the acquired saliency values. Below each image, the used surfel number is shown. This images were derived with the LUTSC-method

a cut-node. In both evaluation strategies is the application of the *TreeCut*-operations negligible.

The advantage of the *Feedback System* is that no predefined LODs are required because they will be defined online using either a priority- or a bucket-approach. The extraction of the priorities is not expensive, and the upload is performed fast. We achieve interactive frame rates on our test systems. A LOD progression is shown in figure 8.4 where the Stanford buddha is reduced from full detail (1M surfels) down to 25k surfels.

As stated before, the DSC has a slower performance, but offers the higher quality saliency values. This is due to the fact that the object does not need to be mapped onto a predefined position. In figure 8.5 we included two pairs, to show the difference based on the saliency calculation methods.



(a) 60k with LUTSC (b) 60k with DSC (c) Difference Image

(d) 60k with LUTSC (e) 60k with DSC (f) Difference Image

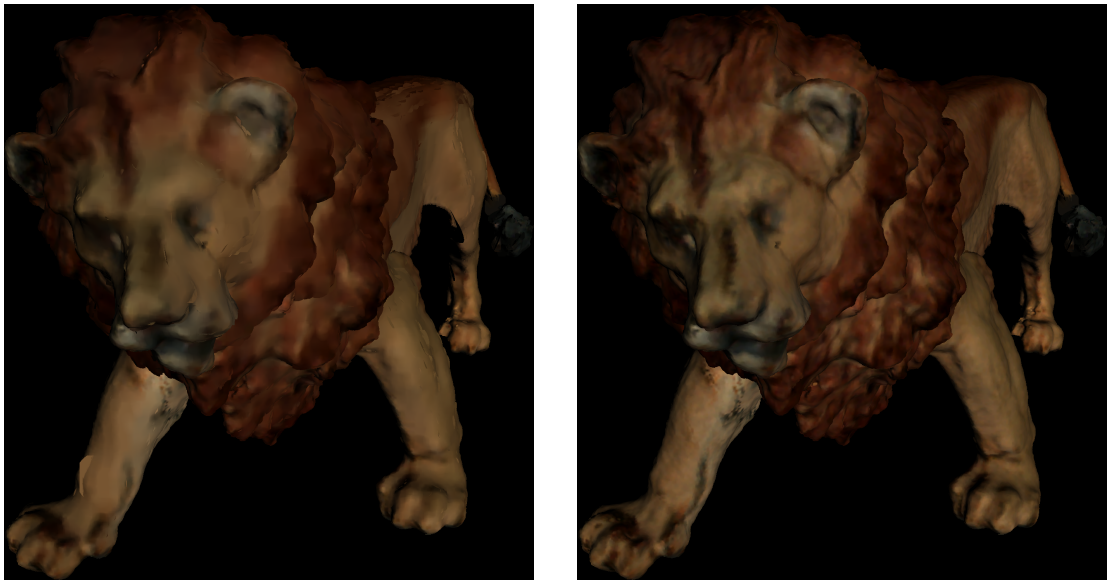
Figure 8.5: Results achieved with the lookup table saliency calculation and the direct saliency calculation-methods in the dynamic feedback method. The top row shows a nearly ideal case for the LUTSC, while the bottom row is a worst case scenario, where a selection of two entries in the lookup table is required. The Standford lion has been reduced down to 60k surfels in all cases. Below the difference image, the used color ramp is shown. The values in each difference image are normalized to the complete scale.

8.4.2 Visual Results

For this work, especially the achieved visual quality is important. Thesis 3 states that the visual quality of the generated results when using the perception-influenced representation is higher in comparison to plain reduction methods. We assume that a better preservation of the visual quality should be detectable when applying both the *Dynamic Data Structure* and the perceptual evaluation.

Two user tests were conducted in which a series of images was shown in a short time span. The time span is assumed to be important as preattentive features are processed in the HVS instantaneously. We assume that the longer the participant is allowed to view an object or image, the more higher cognitive processes will influence a decision. A detailed test design and a complete listing of the results is presented in appendix 'Visual Testing' on page 213. In the following, only the most important findings are presented.

The participants were asked to select the more visually appealing version from a set of two alternative images. They were asked to rate these images with scores ranging from -2 to 2. Thereby, -2 indicates that the first image shown was more appealing, and 2 indicating the opposite case. A value of 0 indicates that either a difference was not visible or no difference was detected. Both are equal for the rating because we presented indistinguishable images. 1 and -1 are tendencies towards an image. The two images shown to a user consists of a highly detailed and a reduced version. The order of the images was random and the time for decision making was logged along with the score.



(a) Low-detailed version (b) High-detailed version

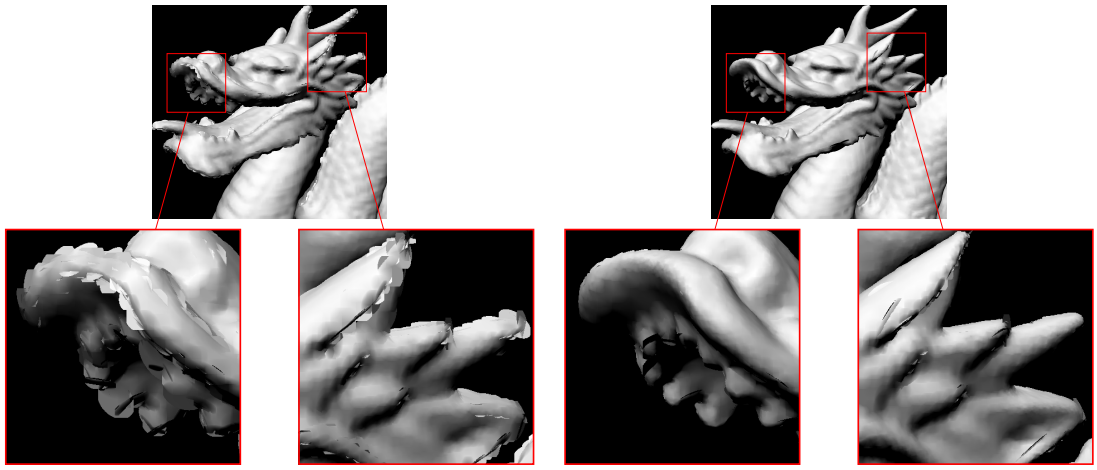
Figure 8.6: A sample test image set presented during the user test. By test design, a participant had only a short time span to select the more visually appealing image.

We displayed a set of images, which were generated with different perceptual evaluator strategies attached to the *Feedback System*. We tested the three reduction methods ("Saliency" (SAL), "Point-Size" (PS) and "Saliency Enhanced Point-Size" (SALPS)), which were presented in section 'Implementation'. Each of these methods uses the existing saliency values differently.

As we use a point-based representation of the objects, the SAL method is expected to be the worst candidate for a reduction. This is due to the fact that point-sizes increase rapidly when the LOD converges towards the root of the tree. The PS reduction method should perform better as a minimal point-size is selected for each cut-node. This way, the complete object's representation is minimized in point-size. The best should be the SALPS reduction method because it additionally preserves details in perceptual important regions. If a triangle-based representation is used instead, the ratings of the SAL method should increase as well.

In figure 8.6, a sample of two comparison images are depicted. For display the Stanford dragon and the Stanford lion were used. The latter includes also per-vertex color information. Thus, it also provides saliency information in non-curvature regions. The dragon object has 1.279.481 leaf points, and the lion is represented with 183.408 leaf points. The object and light positions were chosen randomly during creation of the image set.

We have noted that the size of the objects could be reduced to approximately 10% of its original size without introducing large artifacts. To verify this, a comparison regarding the compression in surfel count for the different LODs had been conducted in a second test. In figure 8.7, a set of images with the PS and the SALPS reduction method is shown. Additionally, the according original object image and a difference image between the two reduced versions are depicted in figure 8.8. More



(a) PS reduction with maximal 40000 surfels.

(b) SALPS reduction with maximal 40000 surfels.

Figure 8.7: Comparison of results generated by the PS and SALPS methods applied for reduction. In image 8.7a, the PS-reduction method has been applied while the SALPS-reduction method has been used to generate image 8.7b. In both images, the objects were reduced to the same number of surfels.

results are presented in the appendix C.

To identify false ratings made by participants, special images were added to the test sets. These were expected to force the user to apply a special rating during the test. We added very bad looking examples where the original object could not be recognized at all. Furthermore, some images were included that did not contain any changes between the low- and the high-detailed LOD-version. In the first case, a higher rating score for the non-bad-looking version is expected while a rating score of 0 should appear in the second case. We used both special image types to identify wrong test results. When such wrong test results were discovered, the complete results were removed before applying the statistical tests.

We tested in total 30 participants in two tests. In the first test series, the display time did not

Test	Participants	Type	Significant [Yes/No]
Score influenced by time	15	linear regression	No ($p = 0.309, R_2 \approx 0$)
Saliency(SALPS + SAL) better than PS	15	t-test	Yes ($p = 0.04648$)
SALPS better than SAL	15	Welch two sample t-test	Yes ($p = 1.2e-06$)
SALPS better than PS	13	t-test	Yes ($p = 0.0003464$)
SALPS / PS score ratio	13	linear regression	Yes ($p = 0.000248$)

Table 8.3: The results of the visual tests. Type indicates the used statistical test method. Significant states whether the results were significant and includes the statistical values as well.

(a) Reference object drawn at full resolution (1.3 M surfels)

(b) Difference between the two reduction methods shown in figures 8.7a and 8.7b

Figure 8.8: Reference of the Stanford dragon and a difference image created based on two result images generated with the *Feedback System*. The difference image is created with the results of the PS and SALPS methods presented in figure 8.7.

significantly predict the rating score ($b = 0.0002$, $t(898) = 1.01$, $p = 0.309 \not\leq 0.002$), and this prediction was also not very good (adjusted $R_2 = 3.824e-05$, $F(1,898) = 1.034$, $p = 0.309$). We assume that these result may be due to a wrong time span used. However, we could not prove our initial assumption regarding the time influence to the rating.

As indicated in the comparison of different reduction methods, the combined version (SALPS) resulted in a significantly higher score than the other methods. In an one-sample t-test, the SALPS reduction methods received a significantly higher score than the PS approach ($t(179) = -1.689$, $p = 0.04648 \leq 0.047$). Both saliency reduction methods performed significantly better than the PS method ($t(387.418) = -4.7771$, $p = 1.2e-06 \leq 0.001$). Because multiple tests were made using the same data set, we applied a Bonferroni correction. The significance value for the complete test has been set to 0.05 while the sub-tests were subjected to 0.002 for the prediction of the score, 0.047 in case of the comparison of SALPS reduction, and 0.001 in the other case.

A summary of results for all tests is listed in table 8.3. These results indicate that the saliency-enhanced LOD techniques do preserve the visual quality of the presented object. The application of the saliency methods lead to better visual results. When using a point-based rendering technique, a point-size reduction method is recommended because otherwise irritating splats are introduced during reduction. With other rendering techniques, the SAL reduction method is assumed to be sufficient. We only used static images, and thus we cannot simply extend our findings to the dynamic scenario. However, as long as the DFM and the *TreeCut* evaluation strategies are processed fast enough, our findings also apply.

In the next test, only the SALPS method was used for saliency-based reduction. The compression in the number of surfels showed a significantly higher score for the SALPS method as long as the surfel count was above 60% of the version that was compressed with the PS-reduction. In comparison to the PS reduction, the SALPS method allows a further reduction of the number of primitives.

There is a linear relation between compression rate and the achieved score ($b = 42.008, t(5) = 9.253$, $p = 0.000248$), and this prediction is very good as $R_2 = 0.9448$ ($F(1,5) = 85.62$, $p = 0.0002477$). This means that the participants were not able to distinguish between the objects even if the SALPS representation had only 60% of the size of the geometrical reduced object. When the surfel count is below 60%, artifacts will become evident, and the rating towards the SALPS method is no longer significant. A graph showing this dependency is shown in the appendix 'Visual Testing' on page 213. As again multiple tests were applied a Bonferroni correction is applied. In this case, all tests are subjected to a significance of 0.016. This results in an overall significance of 0.05 for the whole test.

8.5 Conclusion

The presented *Feedback System* improves the *Model* in our proposed framework by adding the ability of self-optimization. In combination with the BSWDF, a perceptual evaluation of an object is enabled, and reductions in primitive count applied to an object account for preattentive features.

To describe the *Feedback System*, a feedback loop is presented, which includes the *TreeCut*, the *TreeCut*-evaluation, and the data extraction during rendering. All three parts influence each other and generate a highly dynamic system. To increase the stability of the system, a threshold controls the improvements applied by the *TreeCut*-evaluator. When the input data, e.g. the view of the object or the LCMs, is altered and the newly generated priorities are above the threshold, the object is adapted accordingly.

The saliency calculation of the current representation uses the BSWDF and the proposed lookup table approach. The extraction of the features is performed by a shader-program during the vertex shader-stage. We upload the LCMs as a texture. We use the *transform feedback* capabilities of modern graphics cards to extract the results from the GPU back to the CPU. This DFM is only applied if a change of the representation has occurred. The impact of this additional processing is low and interactive frame rates are assured during optimization.

To extract the features during the DFM, either an on-the-fly computation or a mapping of the input primitives' vertices to a sampled LCM is required. The latter utilizes the lookup table capabilities of the BSWDF, which were presented in section ' Approach' on page 113. Interpolation techniques of the GPU allow us to enhance the quality of the conspicuities. We used a linear interpolation of a LCM-Set and a nearest-neighbor interpolation to derive the lookup table entry. Other types are also possible because the LCMs are stored as textures. The BSWDF is applied to the current representation provided by the *TreeCut*. We have established a 1 to 1 mapping between priorities and cut-nodes, and the evaluation strategy directly accesses these calculated priorities.

Additional processing is required to calculate the correct lookup table entry because the stored LCMs have to be matched with the current representation. Therefore, a view is derived that matches to original sampling position based on the view of the object. This view is calculated only if an upload of new LCMs would be necessary. Only then the DFM is activated.

An evaluation regarding the performance of the complete *Feedback System* has been performed showing that interactive frame rates are achieved while the object is extended with self-optimization capabilities. The overhead introduced by the DFM, such as uploading LCM-Set or recalculation of the lookup table entries, is small. Also, the ratio between exchange of new data and rendering calls stays in a reasonable range. Furthermore, the overhead is not generated each frame. Once an object has finished its adaptation to a scenario, only few re-computations are required to optimize the representation to the new scene when assuming frame coherency. Solutions that account for reduction of the LCM size or strategies for uploading the required data efficiently will further increase the performance of the *Feedback System*.

The *Feedback System* has been validated within a user study where a selection of the more visually appealing version had to be made by the participants. The images were presented in a predefined, short time span, and the users were not assigned a special task except for rating the visual quality. We wanted to account only for preattentive features and avoid exhaustive searches in the images. We tested, whether a user is able to make a distinction between low- and high-detailed LOD-versions during this short time period.

The saliency enhanced reduction methods were rated with a higher score than other reduction methods. The achieved results are highly significant. In a second test, the results indicate that saliency-enhanced objects are still favored over geometrical approaches even if the number of surfels is reduced to 60% of the comparison object. However, these achieved results only hold for point-based rendering and static objects. Both mesh-based approaches and dynamic scenarios have to be validated as well. The latter, however, are not affected as long as the *TreeCut* evaluation strategies and the DFM are processed fast enough.

The presented system can be enhanced by several means, and the performance is increased by tuning the *TreeCut* evaluation strategies. As mentioned in the conclusion of the *TreeCut* (see ‘Conclusion’ on page 97), a GPU-based implementation of the evaluation will increase overall performance.

Because of separation between rendering and evaluation, the multi-threading aspects can be enhanced even further by reducing the number of transfers. When the exchange between parallel threads is avoided and waiting times are minimized, an increase of the performance is achieved.

The *Feedback System* and the *TreeCut* evaluations currently account only for a single object. If a full scene with multiple self-optimizing objects is maintained, a different approach for evaluation and data management may reduce overhead. Also, the selection of important nodes could be performed among all objects. For example, this could be achieved with a global priority evaluation and a propagation of the maximal priority values within each *TreeCut* representation.

The transfer of the data may be completely avoided when the entire *Feedback System* is executed on the GPU. With the capabilities of shader model 5, both the compute shader and the tessellation shader provide necessary attachment points for an implementation. When the GPU solely manages the representation, all data transfer is performed in the local GPU memory, which results in a higher performance.

9. Perception-influenced Animation

The presented *Feedback System* is not limited to optimize a visual representation and can be applied to animations as well. In this case, the perceptual information is utilized to identify regions where detail of an animation object can be reduced without creating simulation or visual artifacts. However, the *Feedback System* needs to be extended because an invoked change is no longer restricted to user input. The simulation will change the representation over time and this additional influence needs to be incorporated. We thereby define an animation as the continuous change of an object.

In this chapter we will enhance the presented *Feedback System* to be applicable to animations or simulations, which are provided by an external library. We focus on a single type of animation: a soft-body simulation. Soft-bodies are deformable objects with a restrained surface, i.e. their surface structure is fixed. A common example of a soft-body is a piece of cloth: it is flexible, but the surface is connected through its weaves. The physical properties of a soft-body can be located between rigid-body objects – these are inelastic – and fluids or gases. The latter type is commonly simulated by point-based methods, and these have an $O(N^2)$ time complexity due to the missing topological information. A soft-body based on a Mass-Spring System (MSS) has a time complexity of $O(N+L)$ with N being the number of nodes and L the number of connections between the nodes, i.e. links. During a simulation, i.e. the application of physical laws, the influence of a node to all others needs to be calculated, and thus every reduction will result in a speedup of the simulation.

In this scenario, we include motion and other animation features in the perceptual evaluation. This information can be accessed via the nodes present in the simulation. This way, both surface and animation features are covered by the BSWDF and a reduction in the number of nodes will retain important ones for rendering as well as animation. With the perceptual information, we can account for a human spectator to lower the occurrence of visual artifacts while the simulation remains feasible, e.g. a human spectator would rate it valid. In this context, the individual LODs are referred to as Simulation Levels Of Detail (SLODs) as the detail of the simulation is altered and not the visual representation.

To allow combination of a simulation with the *Feedback System*, the simulation needs to be extended with the *TreeCut*-operations. Also, the application of the *TreeCut*-operations will be adapted, so that a removal or insertion of nodes will not corrupt or invalidate the simulation. For this adaptation, physical and geometric properties are used together when inserting new simulation nodes. These are also utilized to resolve possible generation of artifacts. Thus, a fundamental understanding of physical simulations and constraints solving is helpful, and the reader may be referred to introductory books like [Par08; Mil10].

In the next section, the requirements for a incorporation of a simulation into our framework are stated and discussed. Thereafter, a simulation library is selected, and the necessary adjustments ex-

plained, so that the saliency information can be accounted for. In section 'Extraction of Animation Features', the features and the BSWDF are defined for this special case. After presenting some notes regarding the prototype's implementation, an evaluation of the proposed system is given. The chapter concludes with a summary of the achieved results and further research topics.

9.1 Requirements

As stated before, we aim to establish a combination of a soft-body simulation with the *TreeCut* along with its evaluation algorithms. As in the *Feedback System*, the evaluation algorithm will select important nodes for `refine`- and unimportant nodes for `coarse`-operations. A specialized BSWDF, which includes animation-based features, will be used to derive the relative importance of single nodes.

To avoid unnecessary changes to the *Feedback System*'s design, some requirements are imposed. In combination with an interoperability with the existing *TreeCut* and our framework, the following list of requirements is derived:

- Mapping of *TreeCut* to the simulation structure
- Extraction of features
- Simulation as part of the *Model*
- Physics library for simulation

If these requirements are fulfilled, the *Feedback System* does not need to be changed. The *Model* is still modified via the *Controller* using the data made available during the *Feedback Stage*. The visual output is based on the surface data stored in the *TreeCut*, which can be altered by the simulation. As the design of the *Feedback Stage* is independent of the data, the evaluation does not need to be adapted. However, if a specialized and dynamic threshold for the evaluation would be applied, a more flexible adaptation of an object could be achieved.

The *TreeCut* offers several advantages, especially in the case of a soft-body simulation based on a MSS. The simulation will be applied to the current representation of nodes, and the forces present within the soft-body are computed in the as-is configuration. When neighborhood information is available, e.g. by links, the *TreeCut*-operations can be performed locally. For example, the `refine`-operation stores incident links to the `coarse-parent` and distributes them among the inserted child nodes. No search for these links is required because of the neighborhood information stored. The links between the children can be created without global information as the *TreeCut* provides the parent node during insertion of the children.

The *TreeCut* evaluation strategies operate on the current representation of the *TreeCut*. We therefore require a direct representation of the simulation nodes in the *TreeCut*. An evaluation is then able to select simulation nodes for `refine`- or `coarse`-operations. The BSWDF needs to be extended for the simulation because motion features – presented in section 'Mapping of 2d Features' on page 100

– provide important information and have to be accounted for. In our test scenario, we will only use animation features, but a combination with others would also be possible.

To include a simulation in our framework, the physics calculation need to be incorporated in the used MVC-design. As other objects may not be influenced by the physics calculation, a local design is chosen, which performs these calculations as part of the *Model*. An alternative would be a simulation during the *Controller*-stage as it issues the according updates to the structure. With the *Model*-based update, the representation is solely managed by the *TreeCut*, and the simulation operates on the available data. The *Model* is also extended with global physical properties, e.g. gravity. However, the presented multi-threading approaches need to be adapted. To avoid complications in form of race conditions because of the manipulation of data during simulation, both the *TreeCut* evaluation and the physics calculation will be executed sequentially. In terms of the *Update-Draw* loop, the simulation will be performed during an *Update*-step with an exclusive lock on the data.

Finally, we will use an physics library for the simulation. This not only shows the universal applicability of our framework and *Feedback System*, but also removes the need to validate the correctness of a self-created physical simulation. The physics library may also be more performant, and a parallelization of internal processing steps is achieved without additional effort because it is managed by the library itself.

9.1.1 Related Work

Some research in this area has been done, and some important approaches are presented. Mueller et al. [Mue+04] separate the representation of a surface from the simulation. A set of so called phyxels (physical elements) is used to define the object's behavior while surfels based on a MLS-surface generate the visible output. As this is a point-based approach, no connection between the phyxels is established, and a nearest neighbor search has to be performed prior to each simulation step.

The "adaptively sampled particle fluids" presented by Adams et al. [Ada+07] is a similar method and provides a SLOD mechanism for point-based animation of fluids. By collapsing nodes with low movement and expanding nodes with high movement, more detail is relocated to important areas of the simulation. The authors state that a change of the SLOD should only occur inside of a volume to avoid problems regarding visibility. For display, a MLS-surface is defined, which surrounds the simulation entirely, and the surface is not reduced in size by the SLOD-operations. Again, a nearest neighbor search is required to extract the topological information needed for both the surface and the simulation.

To reduce the accuracy of a simulation during run-time, also multi-resolution representations can be utilized. Beaudoin and Keyser [BK04] proposed a reduction by replacing plant simulations with approximations. These approximations are generated in a preprocess, and the individual parts of a plant can be replaced using a recursive algorithm. To avoid artifacts, smooth transitions are incorporated as well. This geometric approach is both a LOD and SLOD because the detail in the representation and the simulation is reduced.

We propose a similar approach to Beaudoin and Keyser and apply the reduction using a multi-resolution representation. As opposed to the point-based methods, no data separation between the surface and the simulation is required. We assume that the animation is enhanced if a *TreeCut* is used, and its LOD-mechanisms are applicable to simulations. As the *TreeCut* is used, no discrete levels need to be defined in advance and only a multi-resolution representation is required. We also will use a Software Development Kit (SDK) for physical calculations.

The derivation of a perception-based measure when reducing simulation detail has been covered in various user tests and studies [Yeh+09; GDO08; OSu05]. The results of the different experiments show that the precision of a physics simulation can be reduced without loosing plausibility. However, in most of these studies, only the accuracy of the simulations has been reduced. With the help of our proposed system, new studies could be performed, which also account for the removal of simulation nodes.

In the following sections, we will select a SDK, which suits the presented requirements. Afterwards, we explain the extraction of animation features using the mapped *TreeCut*.

9.2 Simulation Library

Different design approaches exist to achieve soft-body behavior, which results from a set of nodes. Usually a MSS or a Finite Element Method (FEM) is applied for simulation. As stated before, neighborhood information between surfels increases the performance of the *TreeCut*-operations. In the case of a MSS, these are given implicitly, but when using a FEM-based simulation, an additional search structure would avoid exhaustive searches during run-time. This search capabilities can also be part of the *TreeCut* multi-resolution representation (refer to section ‘Serialization of the *TreeCut* ’ on page 84).

FEM offers other benefits, such as destruction, plasticity and melting. These could also be represented using a MSS. However, it will have a reduced performance because of the additional maintenance required. Destruction or cutting, for example, is achieved by removing nodes and link information from the active set of nodes. We will utilize the removal capabilities, among others, for adaptation of the simulation to the *TreeCut*. On the one hand, by extending the static nature of the links with a distance attenuation or ripping, a point-based simulation like behavior will emerge. On the other hand, a restriction of the movement will result in a more rigid-body like appearance. Of course, a representation of a rigid-body using a soft-body will not be efficient.

We want to avoid designing and creating a physics library, and therefore need to select an appropriate one. Ideally, not only soft-body, but also rigid-body simulations should be covered by the library. For purely point-based approaches, some SDKs exists, but these are not included within our selection as we will focus on soft-body simulations.

Name	Vendor	Description
nVidia PhysX™	nVidia	Commercial library, GPU accelerated
Havok Physics™	Havok	Commercial library, specialized on real-time collision and rigid bodies
Bullet	-	Open source, collision, soft-body and rigid body dynamics
Open Dynamics Engine	-	Open source, rigid body and collision

Table 9.1: Comparison of suitable SDKs. Both commercial and free libraries are inspected, and a selection is performed. We have chosen to use the Bullet physics engine as it is open source and offers a soft-body simulation. Table derived from [Pal11].

9.2.1 Using a SDK

The list of SDKs supporting the simulation of soft-bodies is growing since the past years. Here, the requirements outlined in section 'Requirements' will be used to select one. The library has to provide access to the internal structure, so that the *TreeCut*-operations are able to directly access the simulation. Ideally, an iterative solver for the simulation is used, which allows to replace simulation nodes without expensive recalculations. The simulation is not required to be physically exact, but plausible. This allows to reduce the computational effort because errors are tolerated up to some degree.

In table 9.1, a list of suitable SDKs is given. All listed SDKs are obtainable via an academic license or are open source. In addition to the library name and the vendor, a short description of the capabilities and features of each SDK is given.

Despite the wide distribution of the first two commercial libraries, the Bullet SDK was chosen because it provides a soft-body simulation and is open source under the zlib-license. This allows to alter the internal source as long as the authors are cited and the license text is not removed.

The soft-body simulation has been added to the Bullet SDK in version 2.68 (we use version 2.79), and it uses a MSS along with a Dynamic Bounding Volume Tree (DBVT) to detect collisions. Local motion of nodes is calculated via the connected springs with an iterative solver, and the global motion is applied to all nodes by adding a constant velocity.

The simulation data is extracted from a mesh-representation, and link information is acquired from the edges. When generating a soft-body object, a patch and an optional set of node masses is provided. In the Bullet physics library, a mass of zero encodes nodes, which are not influenced by any force. These are excluded from the simulation. The springs are initialized with the given set of nodes pairs, and the spring's rest length is set to the distance between each pair. The stiffness of a spring along with other related factors are set during initialization. These configuration variables are defined in advance, but can be changed in a running simulation.

Once the soft-body has been initialized, both the mesh and the masses may be discarded as the simulation will only operate on its internally stored data. For rendering, this data can be accessed, which represents the current state of the simulation. This means that the forces have been applied and

positions and normals have been updated when the data is accessed. The normals are recalculated using the face definition extracted from the initial patch description. If no such description is given, this recalculation will be skipped.

We will now define some typical notions and objects, which are common in physical simulations and the Bullet physics library. The `world` contains all available simulation objects and applies the appropriate simulation model, e.g. a rigid-body simulation for non-deformable objects. An instance of the `world` will be added to our framework, so that all simulation object can utilize this object. The simulation of objects can be disabled and activated to accelerate collision detection or to avoid physical evaluation of unseen objects. All attached objects are accessible for debugging or rendering, e.g. by extracting the required data.

In the `world`, multiple data structures are utilized to detect collisions and to solve physical constraints. This includes a physical `solver`, which updates the object's position in the scene and its physical properties. A `broadphase`-algorithm accelerates searching for collisions, which occur due to the movement of the objects while the `collision configuration` identifies, which collisions are to be handled and what collision resolving strategies are to be applied. The `dispatcher` calls these resolving algorithms based on the involved objects. The `world` additionally provides access to constants, which modify the behavior of the simulation. These include, for example, gravity, wind, or densities of air and water.

The provided `solvers` work in an iterative manner, simulating a fixed time step with an optional maximal number of iterations. This method allows to adapt the numbers of iterations dynamically that increases or decreases the accuracy of the results. The designer or the system has to decide, whether a more accurate solution is required, or a faster, but rougher result is sufficient. The time step determines how much time is simulated within a simulation step. This value is a mandatory parameter in the function call.

The `collision configuration`, the `broadphase`, as well as the `dispatcher` are all collision relevant objects and form the main body of the Bullet library. Acceleration of collision detection is achieved using a `broadphase`-algorithm, which identifies and detects collision objects. The used methods require less than the normal $O(n^2)$ complexity. The complexity of the naïve method is similar to point-based simulation approaches where an object has to be checked against all other. Bullet provides two build-in methods, which utilize search data structures and so reduce the time complexity. The first is a so called `sweep and prune` approach, which exploits temporal coherence in the simulation. The stored information of each object is updated with only a few operations [Coh+95]. The second method is based on a dynamic AABB-tree, which is a specialization of a DBVT. Nodes near the leaves can be updated without the need to propagate the results towards the root. This update also includes swapping and removal. The reconstruction of the bounding volume is iterative and the correct boundaries are approximated fast.

The library supports rigid-body simulation, e.g. objects that are not influenced by inner forces. This type represents the most simple physical object because only gravity, inertia, and collision need

to be accounted for. Bullet defines so called `MotionStates` that allow to provide position information to an object. This way, own data structures can utilize the Bullet library to perform the physical simulation of simple objects. This simulation is enhanced with more complex interaction capabilities by using the `constraints`, which are defined between two objects. Bullet includes several `constraint` types with varying degree of freedom.

It would be possible to construct a soft-body using rigid-bodies and `constraints`, but the `SoftBody`-class encodes the `constraints` directly within the class to avoid overhead. Yet, a special type of `world` has to be generated, which performs soft-body simulation, because it is currently interpreted as an extension to the existing Bullet library. As the `SoftBody`-class maintains a link structure for evaluating forces, this method has a time complexity of $O(N+L)$ where N is the number of nodes and L is the number of links present in the current object. We will include the *TreeCut* into the simulation in a similar fashion, and we extend the existing classes with the necessary features.

9.2.2 Combination with the *TreeCut*

To enable the *TreeCut* to provide the data for the simulation, two possible approaches exist. The first is an implementation of a `MotionState`, which provides the data from and to the SDK. Here the data is accessed by the *TreeCut* directly. The second is to define a new data structure, which extends the `SoftBody`-class.

It is possible to create a rigid-body for each cut-node. Between the cut-nodes, a constraint is generated and maintained. A rigid-body simulation of the complete *TreeCut* is achieved by applying the simulation to the current set of cut-nodes. These positions are propagated via the `MotionStates`. As stated before, this solution does not lead to satisfying results as the modeling of the `constraints` is complex. The overhead because of the great number of rigid-bodies also has a great impact on the performance.

Thus, we adapt the `SoftBody`-class from the Bullet SDK, and include the *TreeCut*-operations along with a dynamic link maintenance. This results in a dynamic LOD for simulations, a SLOD. The fixed structure of a soft-body suits well the multi-resolution representation because a swapping of nodes within this structure is not permitted. This swap cannot occur as the nodes are connected to each other. We have to assure some preliminaries before the simulation data structure can be managed by the *TreeCut* evaluation strategies.

Each node needs access to the incident links, but in the base class, this information is not available. As a selection for `refine`- and `coarse`-operation is local, e.g. no global information is gathered, only the current node is known when applying the operation. To correctly and efficiently reconstruct the link information, the incident links are stored. This avoids exhaustive searches among all links and reduces the time complexity of both operations. Each operation must also retain the current forces applied to the MSS, so that the next simulation step is not influenced by a removal or insertion of nodes. To further reduce any influence, we propose the use of a smooth transition, i.e. a blending between the current and the altered state of the simulation.

The *TreeCut* stores the surfels in the graphics card memory, and thus a mapping from this memory to the simulation is required. We assign each simulation node to a representative surfel when the simulation object is created. This, however, restricts parallel processing capabilities. The simulation cannot be performed during rendering because the surfel data is used in the simulation as well. This, however, is accounted for in the statement of the requirements. In consequence, the simulation is executed during an *Update*-step. As we directly change the surfels in the simulation, no transfer is required. The evaluator, which triggers the *TreeCut*-operations, remains unchanged, but the application of both `refine`- and `coarse`-operations is performed synchronized. No simulation may be performed if the evaluation process is active because the *TreeCut*-operations need to modify the inner structure of the simulation. Due to these restrictions, rendering and evaluation will be performed in parallel while the simulation is performed exclusively.

9.2.3 Updating the Simulation Nodes

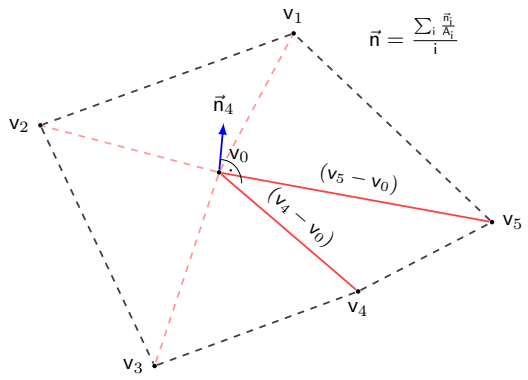
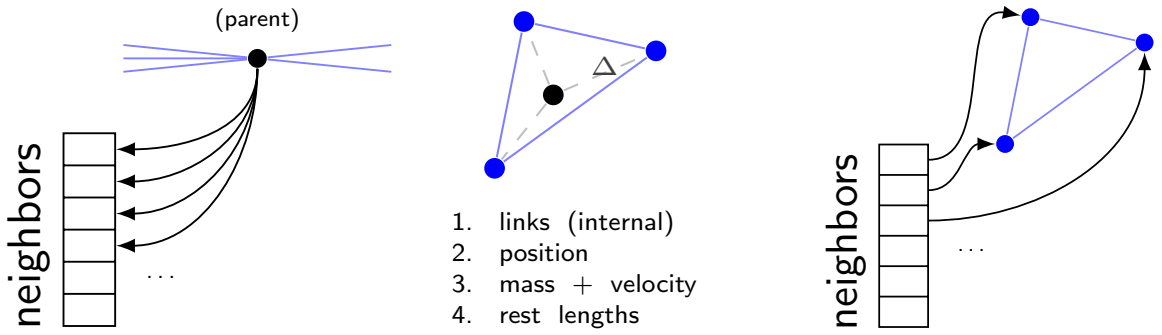


Figure 9.1: Visualization of the face extraction performed when recalculating the surface normal after a simulation step. The links are extracted, and neighboring nodes are used to create a face. This step is repeated for all nodes.

To update the representation of the simulation nodes, the iterative `solver` is called, and the new position information is generated using the current scene and `world`-configuration. As the surface representation may have been changed, the surface normals have to be re-evaluated. This is either done by emulating a face with the links of the represented object or by applying a transformation matrix, which represents the changes applied in the last simulation step. Due to the neighborhood information available via the incident links at each node, we choose to recalculate the surface normals with the help of the first method. In figure 9.1, the selection of the neighboring links and face extraction required for the surface normal's recalculation is shown. To assure correct orientation of the resulting normal, the incident links need to be selected always in the same order. The normal is then derived by a calculation of the cross-product.

Special care has to be taken when replacing nodes in the simulation with the LOD-operations of the *TreeCut*. Not only position information has to be propagated or updated, but also the incident links and physical properties. The `refine`-operation defined by the *TreeCut* replaces the current node with its children. In the initial simulation, the links are distributed among the nodes, so each node is connected with its nearest link neighbor. When replacing a node, the incident links need to be recreated. We store the external nodes, and use a nearest neighbor search along with a minimal incident link count. This minimal link count is necessary because two neighbors may have the same distance and a link needs to be established to each of them. After creation, the children are connected to each



(a) Extract the links that are incident to the parent node. Mark links for interpolation.

(b) Set the simulation parameters. Internal links (between children) are created. Masses, velocities, and rest lengths are calculated. Δ denotes the added displacement.

(c) Reconnect external nodes using the nearest neighbor in the neighbor list. Mark links for interpolation.

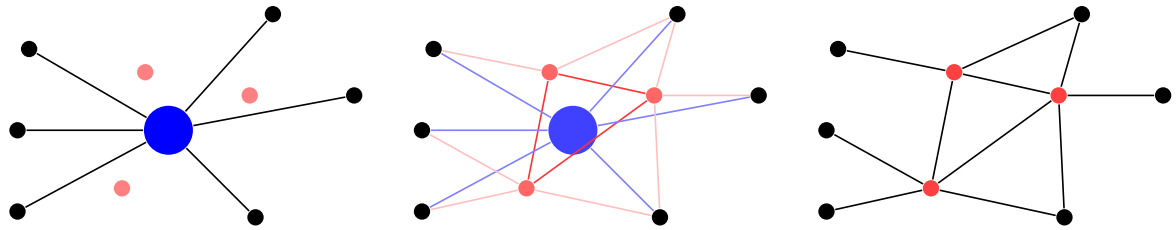
Figure 9.2: The `refine`-operation and the steps required to assure correct insertion of simulation nodes. In the first step, the incident links are extracted. In the second, the physical properties are propagated to the child nodes. In the third, the external links are distributed among the children and interpolation is enabled.

other, so that a certain structure is created, e.g. a quad or a triangle. It is possible to introduce crossing links, but they are not required in our scenario. After this operation, further `refine`-operations will have the correct links available.

In a MSS, no forces exist between two simulation steps – between an iteration they do. The simulation extracts the required information for physics calculations solely from the nodes properties, e.g. previous position, mass, and link information, such as rest length and stiffness. When replacing a node with its children, this information has to be updated accordingly to avoid the introduction of artifacts. The positions of the children are recalculated by adding a displacement value. This displacement is derived from the parent node's position. The mass is equally spread among the child nodes if not stored. The links need also to be changed regarding their constants and rest length. A small example will explain this necessity:

Assume a simulation without any gravitational forces. In the initial set of simulation nodes, the rest length is set to the distance between two nodes. Further assume, a node is replaced with its children, and the rest lengths of the incident links are not updated accordingly. The forces of the incident links will try to move the new child nodes away. The inner links – those among the children – are then compressed and will oppose this force until a new equilibrium is established. The final positions of the nodes would not be equal to the positions defined before the replacement.

Therefore, we need to assure correctness of rest lengths if replacing nodes. This is independent of the *TreeCut*-operation applied. We store the initial positions of the nodes to avoid expensive computations of equilibrium configurations during `refine`- or `coarse`-operations. The rest lengths for the links are then given by the distance between the initial node positions. These are independent of the current ones. However, the additional storage of positional data may not be appropriate, and an



(a) The current links to the node (blue) will be marked for delete and are interpolated, due to the `refine`. The new nodes (red) will be connected with the neighboring nodes.

(b) The influence of the old links (blue) is reduced during the interpolation while the new links (red) gain importance. Inner links are excluded from blending.

(c) The simulation nodes after the applied `refine`-operation. The old nodes and links (marked blue) have been removed and the new links are no longer interpolated.

Figure 9.3: Application of the blending during the *TreeCut*-operations. It is performed to avoid the generation of artifacts. From left to right, a `refine`-operation is performed while it shows the equivalent to the operations performed by a `coarse` from right to left. In the captions, the `refine`-operation is explained.

in-place calculation can lead to equally good results.

Despite the careful calculation and placement of `refined` or `coarsened` nodes, artifacts can still occur as the new positions derived do not match the current equilibrium. We therefore apply a blending between the current and the newly introduced state. The old and new links are marked for interpolation, and the old links are removed once the interpolation has been completed.

In figure 9.2, the replacement of a node is depicted, visualizing the steps for `refine`-ing a simulation node. The mass is set to the sum of the child node masses and the position is extracted from the barycenter of the child nodes. Other physical properties, such as velocity, are propagated as well. In case of a `coarse`-operation, the steps remain the same, but instead of creating internal links, these are removed after completion of the blending. Here again, the interpolation avoids introduction of artifacts. In figure 9.3, this link operation and the blending is visualized.

When adding the displacement of the parent node to the children, the *TreeCut*'s Cut Criterion may no longer be fulfilled[1], or vice versa in the `coarse`-operation. A recovery of this criterion is achieved by either moving the child nodes into the volume, which is defined by the parent position and surfel size, or by expanding the parent's volume accordingly. This displacement may also be wrong with respect to the simulation. With the help of a projection onto the plane defined by the local neighborhood (refer to figure 9.4), more accurate positions may be calculated, and the child nodes can be repositioned accordingly. Note that the rest length is not altered by this operation as this is derived from the initial positions of the nodes.

Due to the projection onto the surface, an approximation is performed, and thus accuracy is reduced when using low-detailed levels. In normal cases, a plain projection defined by the 1-ring neighborhood suffices, but smoother mappings are achieved with the help of MLS or other geodesic projections. These, however, require more computations, and thus need longer time to evaluate the child

[1] Each parent encloses the area of all its children.

nodes' positions.

To reduce space necessary for staring the additional data, a compression of the positions using a delta displacement can be used. In the implementation of QSplat, only 6-bytes are required to store a complete node of a tree, which includes hierarchy information, color, positions, and normals [RL00]. This representation could be adapted and reduced for this special case.

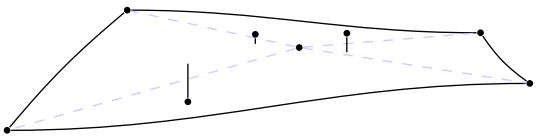


Figure 9.4: Projection of the child nodes onto the local neighborhood plane. Different projection methods can be applied, such as planar or MLS projections.

After a `refine`- or `coarse`-operation has been applied, the newly introduced simulation links are marked for interpolation. This is performed to increase visual quality and simulation stability. After completion of the initial `refine`- and `coarse`-operations, the simulation structure is valid and processing can continue. While the interpolation is active, a blending between new and old links is performed. After the blending has completed, the old nodes and the associated incident links will be removed. As only local nodes are interpolated, the overhead is reduced to a minimum, and in combination with the feedback, an interactive LOD-system for soft-body simulations is achieved.

9.2.4 Integration into the Feedback System

In terms of the framework presented in chapter 'Conception' on page 67, the animation is an additional component of the *Model* (see figure 9.5). The dynamic LOD-methods are implemented via the *TreeCut-SoftBody* (TC-SB). After completion of a simulation step – possibly including multiple iterations –, the data is forwarded to the *Feedback Stage*, and a selection of nodes can be performed. The SLOD-reductions can be triggered by the *Controller* while the *View* extracts the current representation to display the scene.

The *Feedback Stage* can handle both input streams, namely the *View* and the data from the *Model*, without any adaptations. This allows to use both visual and animation features during saliency computations. This is important because in the current scenario time influences the representation of the *Model*, and saliency values will change without a user interaction. The simulation will modify the representation according to the physical properties assigned to the *TreeCut-SoftBody*.

Both processes, the simulation and the change of LOD, require partial or full exclusive lock with respect to the simulation data. Moving regions of exclusive lock allows to overlap these processes. Similar to normal rendering, the processing will also become more efficient when a pipelined computation is performed reusing once calculated data along with SIMD-operations. In the current implementation, rendering and LOD-modifications are performed in parallel while the simulation stalls the others. The implementation of SIMD-operations, however, is left to the physics library and has not been included in our prototype.

A single simulation step (shown in figure 9.6) computes the internal forces of the links and displaces the nodes. The link connection between the nodes restrains the motion and introduces a force

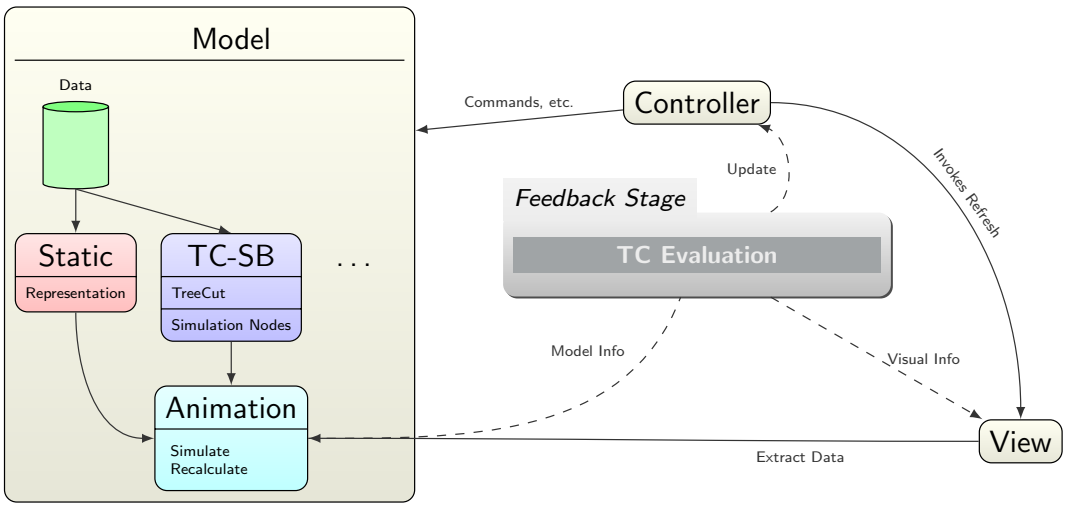


Figure 9.5: The redefined animation saliency framework with the new component in the *Model*. The data is altered by the animation and forwarded to the regular processing steps of the framework. The lines indicate the process loop performed in the system. The *TreeCut-SoftBody* is the combination of a soft-body object in the Bullet library with the *TreeCut*.

to hold them back. Both motion and forces are evaluated over multiple iterations, which splits the movement into several parts. The introduced force are reduced during these iterations. The more iterations are used, the less errors are generated, e.g. overshooting the correct final position. The multiple iterations help to reach a stable configuration, i.e. an equilibrium of forces, faster with the cost of increased computation. After completion of these iterations, the positions are fixed, the forces are cleared, and internal bounding volumes and node properties are recalculated.

The recalculation of the surface features, e.g. the surface normal, can be performed in parallel as the input data is static once the simulation step has been completed. In the Bullet SDK, a method for the recalculation of surface normals is called after the main simulation has finished. By altering this method, the animation features used by the BSWDF are extracted and stored in the surfel structure. This is required because the evaluators access the surfels and not the simulation nodes for evaluating

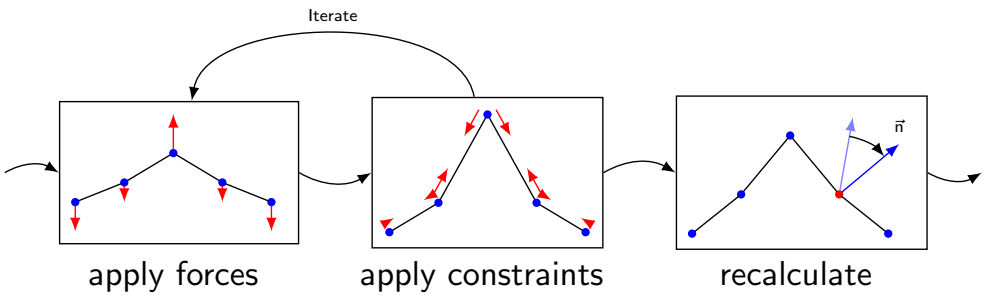


Figure 9.6: The animation loop performed by the SDK simulation including the recalculation of the data. The last step is required for the saliency extraction as this data is needed for the BSWDF.

the priorities via the BSWDF.

During drawing, the representation is adapted, and the simulation is started over. As by design both the *TreeCut* and the simulation operate on the same data, the visual representation is synchronized with the animation.

9.3 Calculating the Saliency Values

When simulating a soft-body, animation features are accounted for by extracting local and global information generated during the iterative simulation. As stated before, these features will be stored in the surfel structure, which can be accessed by the BSWDF during evaluation.

The result is a priority value that reflects the relative importance of a surfel with respect to all other processed surfels. Depending on this priority, the *TreeCut* evaluators are applied and a perception-influenced reduction in simulation detail is performed. If this selection and replacement is applied correctly, the simulation does not introduce any artifacts while the total number of simulation nodes is decreased. This increases the performance of the physics calculation as a linear dependency exists on the number of nodes in a soft-body simulation when using a MSS.

Therefore, it is necessary to identify the appropriate features, which represent the visual salience of an animation. Following this definition, a specialized BSWDF is presented that utilizes these features to compute a single priority value.

9.3.1 Selection of Animation Features

In our case, local and global motion are properties that are directly available in the simulation. These features provide necessary information about the type of modification that has been applied to the object. At least one of these motion types needs to be accounted for if an animation is used. For a definition of the local and global motion, refer to section ‘Mapping of 2d Features’ on page 100. As usual with any feature in saliency calculation, they do not have a dimension, and only absolute values are required. This is because a weighting of the individual features is performed, and after a normalization, multiple features are combined to result in the conspicuities.

The local motion is the relative motion of a node in respect to its previous position, i.e. its velocity. If the simulation moves a single node more than others, this node will have a large value. This matches the detection of motion in the HVS because different moving object catches one’s attention. In this situation, a `refine`-operation applied to that node will increase the accuracy of simulation and the detail of the representation. The overall visual quality will increase because the surface changes in that region are reflected more accurately. In regions where no such explicit node is found, all nodes either move in the same direction, or do not move at all. In this case, a `coarse`-operation will not introduce artifacts in the simulation. A speedup in computation is gained due to the removal of unimportant nodes. A center-surround operation needs to relate the values of a single node to the neighborhood. The result needs to reflect that motions with different direction are also important.

The global motion increases the overall saliency as movable objects are important for the HVS. However, this increase simultaneously limits the ability of focusing on an object. Therefore, the maximal saliency value is clamped to an upper bound that is defined by the velocity of an object (as expressed in spatiotemporal CSFs).

Despite the importance of normal visual salience features, such as color-differences or luminance-differences, the curvature and the silhouette are especially interesting. When the surface of an object is changing, the curvature and the silhouette are altered as well. An initially important region may become unimportant during the simulation as, for example, the surface is flattened. For the same reason, areas of interest may appear and need to be accounted for.

The local motion feature is strongly connected to the generation or removal of curvature values because it indicates where a surface has changed. However, a separated representation is needed because the local motion accounts only for temporal changes. To clarify this, we will give the following example of a cloth that is falling over an object. While it is falling, the local motion correctly accounts for important regions. When the cloth fits itself to the underlying surface, this motion decays. But, the silhouette and curvature are still present. These visual properties contain useful information, which should be included by the BSWDF. Because these are visual properties, the *TreeCut*, and not the simulation, needs to provide the necessary data. Obviously, the lookup table approach (presented in 'Lookup The Data' on page 117) is not applicable in this case, and an online recalculation is required.

We focus on the compatibility of the new system to our existing framework. Therefore, we omit any visual features and account only for the animation ones. Thus, the following definition is solely based on the simulation data, but an extension with visual features is still possible.

9.3.2 Animation BSWDF

The BSWDF operates on both local and global motion features. As stated in the definition of the BSWDF (refer to 'The BSWDF' on page 101), global and local features are separated because the global features influence the local ones. Their individual sums are combined to result in the animation saliency. In this special case, only two features are included. Thus, the following, reduced definition of the animation BSWDF is derived:

$$\text{BSWDF}(\omega_C, \omega_L, d_C, \vec{x}) = \text{Illu}(\omega_L, \vec{x}) \circ \text{Animation Features}(\omega_C, d_C, \vec{x}) \tag{9.1}$$

with ω_L being the light, ω_C the camera in spherical coordinates, and d_C the distance to the camera. We define the Animation Features as

$$\text{Animation Features}(\omega_C, d_C, \vec{x}) = \text{Global Motion}(\omega_C, d_C) \otimes \text{Local Motion}(\omega_C, d_C, \vec{x}) \tag{9.2}$$

The position $\vec{x}$ is the current surfel's position assigned to a simulation node, which has been iteratively updated during multiple simulation steps. If no illumination is used, the BSWDF simply

evaluates to the result of the Animation Features. Like in the BSWDF definition, $\otimes$ is the operator that limits the local features based on the global properties of an object.

The BSWDF is computed after the final stage of the animation loop, i.e. the recalculation of the surface properties. Then, the most current representation of the simulation nodes is available, and the BSWDF can evaluate the most current saliency values. The cut evaluation is performed in parallel to rendering, which invokes the `refine`- and `coarse`-operations on the *TreeCut-SoftBody*. This is possible because the rendering of a *TreeCut* uses an index-list for rendering, and the operators only modify a copy of this index-list.

If the rendering algorithm additionally provides visual salience features, these can also be accounted for by the BSWDF. Both the animation features and the visual features are present during computation. For example, the proposed GPU-method (refer to section 'Calculation on the GPU' on page 110) could provide the necessary priority values for each surfel. The *Feedback System* then accounts for both visual and animation features simultaneously and adapts the soft-body accordingly.

9.4 Implementation of the Animation

The simulation of the soft-body is strongly connected to the *TreeCut Dynamic Data Structure*. As rendering is not manipulated or altered by any means, a mapping between cut-nodes and simulation nodes is required. This mapping also includes access of the surfel data, which is managed by the *TreeCut*. Furthermore, an identification of a simulation node using a cut-node is necessary to keep both data structures synchronized. Only a cut-node is used as parameter when the *Controller* issues a `refine`- or `coarse`-operation.

We closely followed the design of the Bullet SDK to avoid having to make changes to the physics library as it could change with the next version, and extended it with the necessary functionality. A clone of `SoftBody`-class is created and included in our framework, the *TreeCut-SoftBody*. In the following, the necessary extensions will be presented allowing our *TreeCut-SoftBody* to simulate a soft-body using the *TreeCut* data. This performs a dynamic SLOD, which is controlled by our framework. The transition is made in four steps, which represent the individual stages within the generation or simulation of a *TreeCut-SoftBody*:

- Mapping and extension of simulation nodes
- Generation based on an existing *TreeCut*
- Application of the dynamic methods
- Recalculation of animation features

9.4.1 Adaptation of the Simulation Nodes

The `SoftBody`-class creates an internal structure for the simulation nodes. This includes, besides position information, properties like force, velocity, and mass. We want to avoid filling the graphics

card with unnecessary information, so this data remains separated from the rendering. Yet, this results in the need to identify surfels in the *TreeCut* not only via a cut-node, but also via a simulation node. We utilize a hash map to connect a surfel with a node, which enables a lookup of the assigned node if a *TreeCut*-operation is triggered. A simulation node stores the index of its assigned surfel to allow modification of the surfel position. When replacing, deleting, or moving a node in memory, this information must be updated.

During a *TreeCut*-operation and recalculation of surface properties, local connection information is helpful to avoid exhaustive searches. We therefore use an incidence list, which stores all links that are connected to the current node. In the MSS, this is not required because all links need to be processed sequentially, and an individual lookup is not necessary. For the dynamic LOD, however, this accelerates recalculation as the local links are directly available.

During a replacement or a simulation step, the position information has to be accessed. We therefore map the *TreeCut* surfel buffer to the nodes, so the simulation directly operates on the surfels. This way, a copy of the positions for each node is avoided, but then the rendering may not be performed during a simulation step. Additional physics properties, such as previous position, mass, etc., are still stored in the internal node structure.

To reconstruct the correct rest length of a link, we store the initial positions of the *TreeCut*'s surfels. This increases the memory requirement of the simulation, but the rest lengths are exact, and this avoids costly calculations. Other methods may overcome this limitation and circumvent the storage of the initial positions.

Based on this enhanced node structure, the *TreeCut-SoftBody* is generated, and the dynamic methods can access the needed data. The dynamic methods are executable with only local information, e.g. a simulation node. Additionally, the necessary information for the dynamic methods is assigned to the *TreeCut-SoftBody*.

9.4.2 Generating a Simulation Using the *TreeCut* Data Structure

In the Bullet SDK, a soft-body object is generated with the help of an initial patch or object from which the required information is extracted. This method is adapted to a *TreeCut*, and the connections between the simulation and the rendering data are established.

When initializing the *TreeCut-SoftBody*, the *TreeCut* is iterated, and for each cut-node, a local simulation node is created, which is thereafter used by the iterative solver of the *TreeCut-SoftBody*. If masses are available, these are stored as the inverse for further calculations in each simulation node. Otherwise, a fixed, predefined mass is used. As in the `SoftBody` class, a DBVT is generated and maintained. When creating the local node, the mapping between surfel index and simulation node index is stored in the hash map.

For the generation of the incidence list and the global link list in the *TreeCut-SoftBody* object, a nearest neighbor search can be performed. This step would be required only during the initialization. We, however, follow the `SoftBody`-class, which manually identifies the needed connections. This

way, only necessary links are generated. Also, a face definition can be extracted as well if wanted. To extract point-sizes of the surfels, one can choose to select the longest incident link or to create a NNG. The latter method is far from being efficient, and if a mesh or patch is available during the generation of the soft-body, its edges should be used instead.

After initialization, the simulation of a *TreeCut-SoftBody* can be performed. The links provide the necessary surface connections, and collisions are handled correctly due to the DBVT structure. Additionally, environmental effects, e.g. wind, can be applied, too. However, to simulate volumetric objects, which are covered by the *TreeCut* as well, some more details have to be accounted for.

Notes on Volume Preservation

If a volume is given, the link structure does not guarantee compression effects to be resolved correctly because only the surface structure is restrained by the links. For example, when gravity is applied, the links are not able to restrict the motion of nodes, and the object will start to collapse. More formal, the volume will be compressed or destroyed, which may not be the desired behavior.

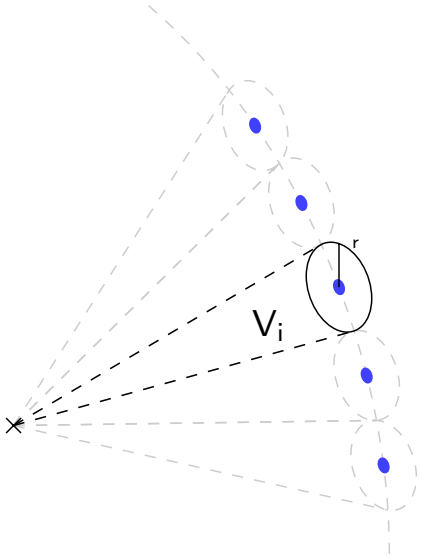


Figure 9.7: Volume preservation for point-based representations. The volume is approximated with a cone from the barycenter of the object to each surfel. The base radius is defined by the point-size r.

Basically, two options exist to circumvent this compression of the volume. Both try to preserve the volume of the provided object and apply the forces accordingly. These forces arise due to compression or expansion of the object when performing a simulation step.

The first method creates links to retain the inner structure of a soft-body. These links prohibit a collapse of the shape, but a large number of links is necessary. The links need to be generated especially to keep the shape, and thus more calculations are performed. In case of a convex object one may achieve this by inserting a simulation node in the barycenter of the soft-body creating links from the center to all surfels. With these links, the volume can be controlled in its behavior, e.g. to create a soft or hard volume preservation. As the shape is convex, no crossing links are introduced. If a more complex object is used, this method fails. We therefore propose a different, more universal approach.

The second method is to create a force in direction of the surface normal of each node when a change to the volume of the object occurs. The greater the compression or expansion of an object, the greater the displacement force. The change in volume is signed, and thus an expansion leads to a force that shrinks the volume, vice versa for the compression.

In the Bullet physics library, the volume of a soft-body is approximated by summation of tetrahedra volumes. Each tetrahedron has its base at an distinct origin node in the set of all simulation nodes.

Then, with all faces that describe the surface of the soft-body, the tetrahedra are generated, and their volume is calculated using the formula:

$$V(x, f_{v_1}, f_{v_2}, f_{v_3}) = \frac{|(f_{v_1} - x) \cdot ((f_{v_2} - x) \times (f_{v_3} - x))|}{6} \tag{9.3}$$

where f_{v_i} denote the individual vertices of a face extracted from the initial patch or mesh description. x denotes the base vertex.

This is not applicable to the *TreeCut-SoftBody* data structure as in the *TreeCut* representation faces do not have to exist. Instead, another solution is proposed using only properties available from the surfels.

The volume is approximated by cones instead of tetrahedra because these can be defined using the available surfels. As in the Bullet SDK, a base surfel is selected, and each node provides the end position and the radius of the assigned surfel. This is shown in figure 9.7. The radius is extracted from the surfel using the parent's boundaries, i.e. the point-size. The cone volume is calculated using the surfel's position p_s, its radius r_s, and a center point p_c as follows:

$$V(p_s, r_s, p_c) = \frac{1}{3}\pi r_s^2 \|p_s - p_c\| \tag{9.4}$$

By summing up all cone volumes, the object's complete volume is approximated, and it can be compared with the volume calculated in the previous simulation step. If an object is compressed, i.e. the volume gets smaller, the surfels are extruded along the surface normal. If the volume expands, the nodes are drawn back into the original volume. Note that only the volume and not the shape is retained by this method.

The most accurate results regarding the volume calculation are achieved when the barycenter of the object is used. As only a comparison with the previous volume is performed, a randomly chosen surfel suffices as tip of the cone, like in the original `SoftBody` class.

9.4.3 `refine`- and `coarse`-Operations

The `SoftBody`-class allows cutting of nodes within the simulation. This operations removes all incident links from the selected set of nodes. This operation is similar to a `coarse`-operation, but the complete removal of the nodes is not desired. Furthermore, this operation iterates all existing links to find incident ones. We enhance this method and also allow insertion of nodes into the simulation data structure. This way, both `refine`- and `coarse`-operations are implemented in the *TreeCut-SoftBody* object.

The links must be redistributed to recreate the original structure of the surface. Thus, the incident links to or, more precisely, the neighbors of a node are stored and reconnected after an operation has been completed. Otherwise, the topology of the object would be altered, and the simulation would be different as it would be influenced by this change.

Starting the *TreeCut*-Operations

In the following, w.l.o.g. a `refine`-operation is assumed to be performed. As stated in section 'Updating the Simulation Nodes', the incident links and the node properties are propagated towards the children. Therefore, the following steps need to be conducted:

1. Store incident links
2. Propagate properties
3. Create incident links
4. Mark for interpolation

When `refine`-ing a node, the incident links are marked for interpolation, and adjacent nodes are extracted. These are stored for later use because they are required for reconnection when the children have been inserted. We store the nodes instead of the links as the new incident links are created using these nodes, and no further extraction is then required. The nodes are also needed to assign the correct properties to the newly introduced links.

The physical properties of the parent node are acquired and propagated accordingly. The mass is divided by the number of children present while the velocity is not scaled. This is the case because the applied force to a node is evaluated with the formula $F = m \cdot a$. A division of the velocity would result in a different force as the acceleration is computed using the current velocity. Afterwards the child nodes are added to the set of simulation nodes, but their bounding volumes are not included until the parent node has been removed, which will done when the blending is completed.

The position information, i.e. the previous and the current position, of the parent node is handled in a special way. A displacement of a children node relative to its parent is calculated based on the initial positions. This displacement is then applied to both positions and is stored in the node. An approximation is performed due to the fixed positioning, and more elaborated methods might give more exact results. The presented projection method should increase the quality of the computed positions. It is also possible to place the child nodes in a way that no forces are applied to the MSS.

After propagation, the adjacent nodes are connected with the new nodes. A local nearest neighbor search is performed among adjacent nodes, but this time a brute force method is used. As the number of children is known, we use a fixed distribution to create links between them. The adjacent nodes are connected in a first-come first-serve order. The created links are added to the incident link lists of both nodes and are marked for interpolation. As in the initialization of the simulation, we do not create crossing links between the individual children. The rest length of the new links is, again, extracted from the initial positions stored during generation. Note that in our prototype, the maximal number of children is set to four, and no crossing links are generated between them. However, if a larger number is given, a more universal approach may be required.

Ending the *TreeCut*-Operation

After completion of the interpolation, the old nodes and links are removed from the *TreeCut-SoftBody*. To safely remove the links, their internal order may not be altered. Otherwise, a recalculation will lead to different positions of the nodes. This is due to the fact that the links are evaluated sequentially and changing their order affects node positioning. After several iterations, a stable configuration would be obtained again. However, during this period artifacts will be introduced, and therefore we state the following steps to end a *TreeCut*-operation:

1. Finalize physical properties of the new nodes (e.g. insert collision shapes)
2. Delete old links in sequential order
3. Delete old nodes

The physical properties of the newly introduced nodes are updated or finalized. In our case, the collision shapes are updated and inserted into the DBVT. This step is done after the old simulations nodes have been removed from the DBVT. Also, the incident links of the newly introduced nodes are no longer being interpolated.

Before the old nodes are removed, their incident links are used to extract the affected links that are to be deleted. As stated before, the order in the set of all links must be retained. Therefore, a linear traversal is required. This increases the theoretical time complexity of the end algorithms from $O(1)$ to $O(L)$ where L is the number of links. This has no large impact as it is possible to use a lazy deletion strategy to reduce this complexity. As the links are processed when calculating the forces, the links that were marked for deletion can be identified, removed, and skipped. This does not introduce additional costs. Finally, the simulation nodes that are no longer required by the simulation are removed. As the nodes are order independent, a deletion can be executed in $O(1)$.

9.4.4 Extraction of Animation Features

The recalculation of surface properties and the animation features is performed after a simulation step is completed. In this phase, the nodes are fixed and the complete bounding volume has been updated. The recalculation is done as described in section 'Updating the Simulation Nodes'. This step can be accelerated by parallelization because neither the evaluation nor the physics engine require access to the data.

To recalculate the surface normal, a plane based on the local neighborhood is created. This neighborhood is implicitly given by the list of incident links. Once the neighbors are available, a surface normal is recalculated by computing the cross product of the vectors between the source node and the nodes in the neighborhood. The resulting vectors are summed, normalized, and stored.

During the extraction of the surface plane, local information from all nodes is available. This is used to extract the relative motion of a source node, i.e. its velocity, is extracted. The global motion is

derived by summing up all local motions. As stated in section 'Calculating the Saliency Values', only absolute and dimensionless values need to be stored.

All recalculated data is stored in the surfels of the *TreeCut*. This assures that the BSWDF has access to this information when determining the priority of a cut-node. If no further information for a BSWDF is required, an in-place computation of the saliency values may accelerate evaluation and reduce the memory footprint of the surfel buffer.

In the normal case, the visual and animation features are provided to the rendering method. Based on the available data, the BSWDF is applied, and the priorities of the rendered cut-nodes are extracted. For this reason, a storage in the surfel attributes is required. Otherwise, no access to the animation features would be possible. If no animation is applied or no animation-specific data is available, the animation features are set to zero. Therefore, the BSWDF evaluation is fully independent of the *Model*.

9.5 Performance of the Proposed System

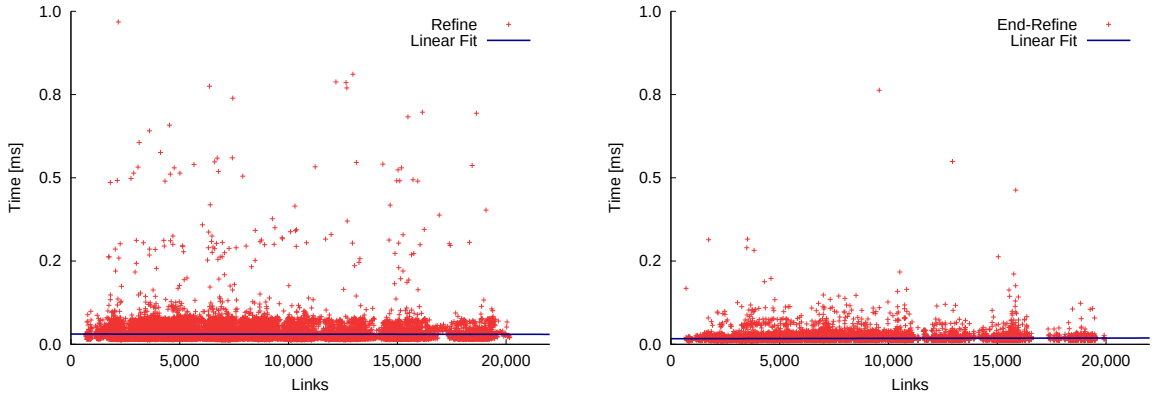
We have implemented a prototype of the proposed *TreeCut-SoftBody* in `C++` and included it in the existing framework. For adaptation of the representation, the priority-based *TreeCut* evaluation strategy is utilized. In the following, we will present some performance results that have been achieved using the proposed animation system. We will focus on the performance of the individual *TreeCut*-operations that have been mapped.

A cloth object is generated using a regular grid, and with this the required LOD hierarchy is generated. The internal nodes, which are extracted from the *TreeCut* representation, have a mass of 1 in the initial configuration. The corners are fixed in the simulation, i.e. their mass is set to zero, to create a swinging net. In the presented tests, only a gravitational influence on a *TreeCut-SoftBody* object is simulated. All presented tests were performed on test system 1, its configuration is shown in appendix 'System Configurations' on page 211.

In the first test, we measured the performance of the dynamic SLOD-methods. This test is important because the time required for simulation step is meant to be reduced in comparison to the `SoftBody` object in the Bullet library or the original version, i.e. whose detail has not been altered. In table 9.2 the manipulation of a single node is presented, and these results are averaged over multiple applications of the *TreeCut-SoftBody*-operations.

The results of both *TreeCut*-operations show that the overall performance of the operations is high. An interpolation is performed that blends between the affected nodes and requires a cleanup after finishing. Therefore, the according end-operations are included, too. Each entry in the table contains the fraction of the complete processing time along with its measured times. As expected, the removal of the incident links after finishing the interpolation is constant. This dependency is visualized in figure 9.8 with the `refine`-operation. The same findings apply in case of a `coarse`-operation.

In the second test, the simulation time with respect to the number of simulation nodes and links is



(a) The `refine`-operation

(b) The end-`refine`-operation

Figure 9.8: The measured times of the `refine` and end-`refine`-operations applied by the *TreeCut-SoftBody*. A linear fit is shown that uses all samples generated by an operation.

Operation	N	L	Node-Ops [ms]	Link-Ops [ms]	Complete [ms]
`refine`	4888	10419	0.0164 (91.14%)	0.0016 (8.94%)	0.0180
end-`refine`	4889	10794	0.0017 (94.39%)	0.0001 (5.53%)	0.0018
`coarse`	4966	10615	0.0319 (97.61%)	0.0007 (2.39%)	0.0327
end-`coarse`	4938	10917	0.0025 (99.20%)	0.000 (0.79%)	0.0025

Table 9.2: Different timing results when applying the *TreeCut*-operations to the *TreeCut-SoftBody*. N denotes the average number of nodes present while L is the average number of links. The Node-Ops are all operations that affect a node of the simulation while Link-Ops modify its links.

evaluated. Therefore, a test set is defined consisting of multiple nets with varying number of nodes, which ranges from 400 to 10000. In the **NoReduction** case, the plain simulation is performed without any interaction of the *TreeCut*-operations. In the other cases, a reduction is applied with varying compression factors, i.e. 90%, 75%, 50% and 30%. A simulation is considered completed after a fixed iteration count – we used 1000 iterations. During each test, no collisions are performed. All methods are independent of their compression rate and start with the same initial number of nodes. For all tested objects, the recalculation of the surface normal is excluded from the results.

Table 9.3 shows the averaged timing results over a test set while table 9.4 contains the distribution of the processing among the individual operations performed. The result indicate that a reduction in the number of simulation nodes is worth the additional computations. The SLOD-reduction accelerates the simulation. As it was shown in the previous test, the application of the *TreeCut*-operations does not have a large performance impact. Thus, the overhead generated due to the evaluation is amortized. During a single simulation step in the Bullet library, a “default single step” (DSS) is called for all objects. This is followed by the constraint solving (CONST), and a call to feature recalculation and interpolation (R+I). This interpolation, however, contributes to the overall process only if a `refine`- or `coarse`-operation has been applied.

Method	Reduction	N	L	I	Total [ms]
Bullet SoftBody	-	4266	8413	0	11.09
TreeCut-SoftBody	NoReduction	4266	8647	0	10.08
	90%	3901	8004	22.0	9.62
	75%	3324	6931	30.5	8.40
	50%	2492	5472	40.6	6.32
	30%	1911	4220	25.16	4.80

Table 9.3: Timing results with and without the feedback loop using varying node counts. The Reduction is the factor how far the simulation is being reduced. N denotes the node count, L the number of links present and I the number of interpolations. All results are averaged over a complete test set.

Reduction To	DSS [ms]	CONST [ms]	R+I [ms]
NoReduction	4.995 (49.54%)	4.929 (48.89%)	0.156 (1.54%)
90%	4.737 (49.21%)	4.662 (48.43%)	0.225 (2.34%)
75%	4.083 (48.59%)	4.084 (48.60%)	0.235 (2.79%)
50%	2.922 (46.19%)	3.170 (50.10%)	0.233 (3.68%)
30%	2.226 (46.33%)	2.422 (50.39%	0.156 (3.26%)

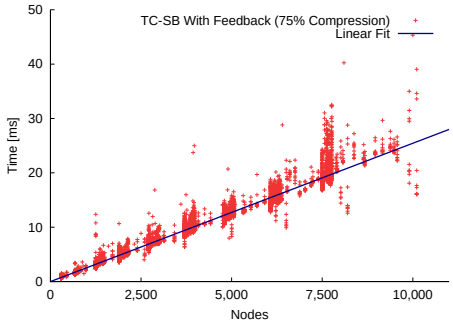
Table 9.4: The average processing times of the individual steps performed during a simulation step of the *TreeCut-SoftBody*. In addition, the fractions of each process is given.

The *TreeCut-SoftBody* decreases the number of simulation nodes of the object and increases simulation performance. In the averaged results, which were extracted after completing a test set, a reduction in processing time to approximately 47.76% is achieved in the best case (reduction to 30%). In this case, on average 25 additional interpolations are performed each simulation step. The default `SoftBody`, however, is not faster. Additionally, it is not possible to alter the detail using the `SoftBody`. The only option would be the re-initialization with the new configuration, which would not be efficient at all.

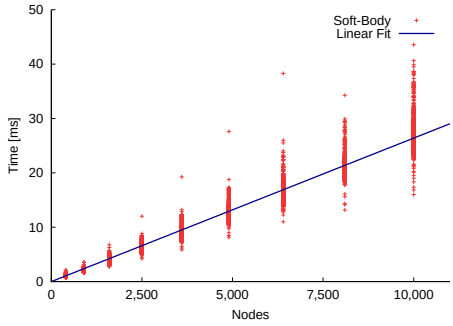
The impact to the simulation, which includes the feedback and the interpolation, is rather low. In figure 9.9, the results with dependency of the node count are visualized. As expected, a linear time complexity is achieved, and with the reduction in node count, a faster processing is possible.

In figure 9.10, some visual results achieved with the *TreeCut-SoftBody* are shown. In this figure, a comparison with the Bullet `SoftBody` is performed, and a fixed time step is used to generate the results. For reconstruction of the surface, we did not use the *TreeCut* splatting methods, but an extraction of faces using the incident links of each node. After several iterations, some artifacts may occur, which corrupt the boundaries of the cloth (see figure 9.10h). In each row, the images are taken after zero, 45, 90, and 180 simulation steps using a default view above the object.

In the shown images, we only used the local motion features for BSWDF evaluation as the global motion is zero in this scenario – it is a fixed object. We prevent application of a `coarse`-operation at the boundary nodes, i.e. those at the edges of the cloth. For this reason, the edges may not be reduced in size, and artifacts due to wrong reduction are avoided in these regions. The results show that the



(a) *TreeCut-SoftBody*-simulation with Feedback. Here, a reduction to 75% to the initial size is applied.



(b) `SoftBody`-simulation without Feedback

Figure 9.9: The performance of the simulation in dependency of the node count used. A comparison between the `SoftBody` and the *TreeCut-SoftBody* is performed. Despite the additional calculations, the *TreeCut-SoftBody* is faster. The results are taken from a test set.

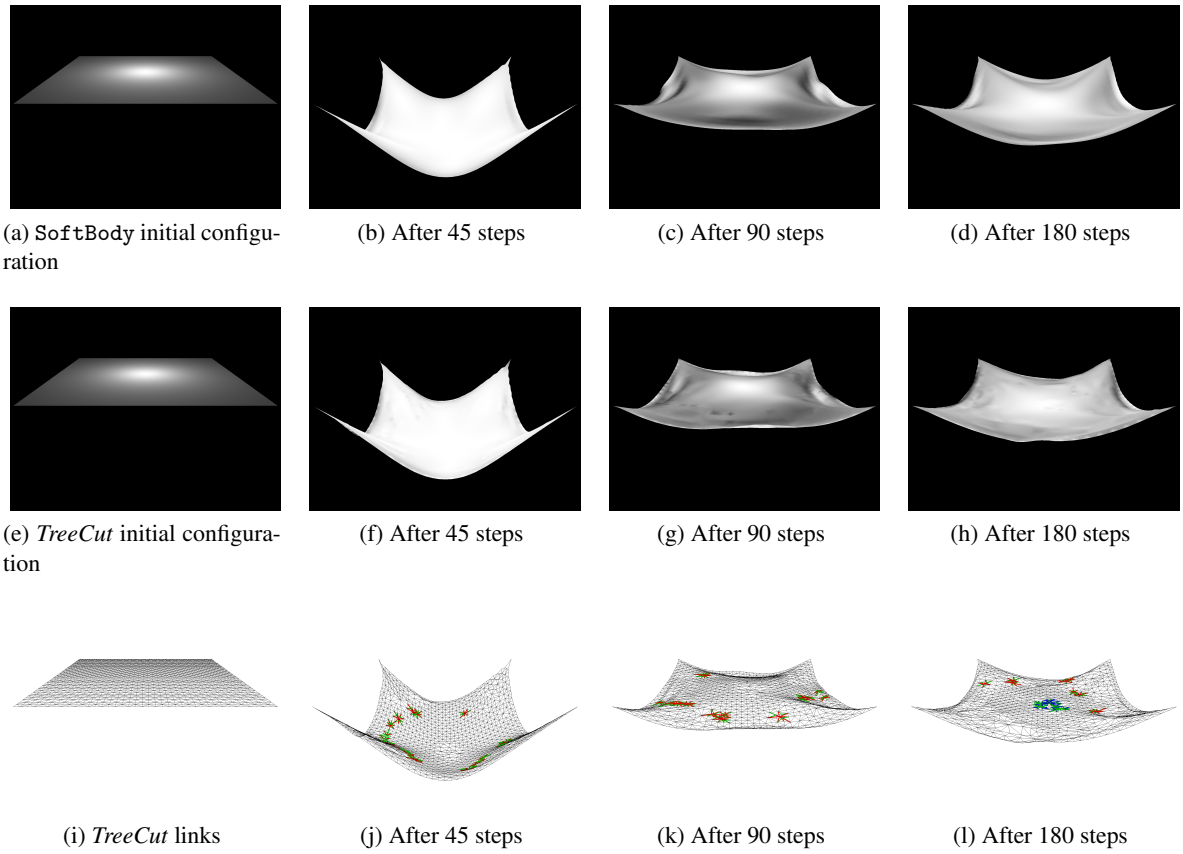
reduction preserves the inner detail of the object. The newly introduced links and nodes only generate small errors. Especially in the fast moving center region, most detail is present because almost no `coarse`-operations have been applied there. This fast moving region has been correctly identified by the BSWDF. In figures 9.11a and 9.11b, are more detailed view of the distribution of links can be seen.

9.6 Conclusion

The presented system extends our proposed *Feedback System* and simulates a soft-body object. A SLOD-reduction is applied based on visual salience, and for calculation, specific features for animation are also included. With the proposed animation system, the number of simulation nodes can be reduced online without any definition of discrete levels in advance.

Unlike full point-based approaches where such a SLOD system has been presented before, the removal of simulation nodes may not be performed without preconditions. The soft-body simulation, as defined by the Bullet physics engine, requires to maintain a link structure between the surfels to evaluate the applied forces. The naïve removal of a link is only one piece of the puzzle to modify the SLOD of a simulation. We complete this by storing and extracting incident links of each node and provide these to the *TreeCut*-operations. The links are then recreated in a local neighborhood. This enables dynamic adaptation of detail using the *TreeCut*. As the animation system is part of the *Model*, no changes are required to allow our *Feedback System* to enhance the representation.

We defined a specialized BSWDF, which operates on animation features, such as local motion. The saliency values can be computed either during rendering or within the simulation. These are then forwarded to the evaluation. From that point on, the normal processing is continued as presented before.



(a) `SoftBody` initial configuration (b) After 45 steps (c) After 90 steps (d) After 180 steps

(e) *TreeCut* initial configuration (f) After 45 steps (g) After 90 steps (h) After 180 steps

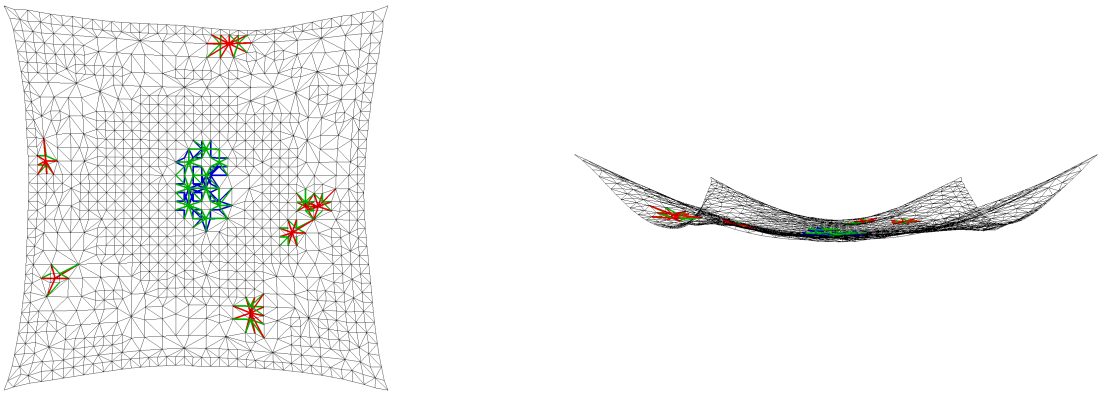
(i) *TreeCut* links (j) After 45 steps (k) After 90 steps (l) After 180 steps

Figure 9.10: Comparison of results generated by the `SoftBody` and the *TreeCut-SoftBody*. A fixed time step is used to generate the different configurations. In the last row, the links of the simulation are visualized. The highlighted links are being replaced by the *TreeCut*-operations. In the image (h) the artifacts of the boundaries occur because of a large reduction was performed. Also, illumination artifacts are generated by the surface normal calculation method.

Currently, only one *TreeCut* is applied for both rendering and simulation. Other methods, e.g. as presented by Mueller et al. [Mue+04], are based on two independent representations. This allows to reduce the number of simulation nodes independently of the surface representation, but a second data structure is introduced, which seems unnecessary. We plan to apply a second *TreeCut* instead, which represents the data for simulation, and thus only one underlying data storage is required. It is necessary to propagate the results of the simulation, for example, during the final update when animation features are extracted. The surfels used for the rendering by the *TreeCut* are then updated accordingly.

When applying a *TreeCut*-operation, the placement of new nodes is performed using only geometrical measures. This needs to be extended with animation and physical measures as well. This will increase the stability and feasibility of the simulation. The generation of a smooth surface would help to reduce artifacts during these operations. A MLS-surface seems suitable for this case, and the new simulation nodes can be projected onto it.

The proposed method only applies for surface-based simulations, but the *TreeCut* itself is not



(a) Top view of the reduced *TreeCut-SoftBody*

(b) Front view of the reduced *TreeCut-SoftBody*

Figure 9.11: More detailed view of the link distribution after applying 180 simulations steps. The size of the net has been reduced to 300 from initially 400 nodes. As local motion is evaluated, nodes near the center are not reduced.

limited to a specific representation type. For example, a *TreeCut* is able to maintain volumetric representations, too. The question is what changes to the presented animation features and to the simulation are necessary when using volumetric approaches, like FEM-based simulations.

A GPU implementation could increase performance because it allows efficient computations. Additionally, if the data is directly available on the graphics cards memory, no mapping is required. With the capabilities of the newest shader model 5, a soft-body can be implemented without any restriction. It remains to be seen how the proposed animation system needs to be updated and how the *TreeCut* can be included in the graphics card.

The complete animation system accounts for animation features, and thus includes perception during the evaluation of the simulation object. However, user tests have to be performed to validate the gained impression of presevered detail. Also, it needs to be shown that the simulation remains plausible even if nodes are `coarsened` or `refined`. A maximal compression factor could be derived to identify when a simulation looses its correctness or plausibility.

10. Conclusion

The integration of perceptual information within computer graphics and especially in 3d rendering provides useful applications. The ability to model the human visual pathway enables us to exploit limitations of the visual system in rendering scenarios. These can range from simple reductions up to steering the user to perceive specific information of a scene by looking at it. However, until now no method was available that could utilize perceptual information effectively. In this PhD thesis, we implemented a perception model operating on 3d data, and defined a dynamic data structure to smoothly apply changes to an 3d object. Unlike previous methods, we operate on the rendering pipeline and achieve interactive adaptation of an object.

Our approach uses an early vision model, so no external hardware components are required to evaluate an object. This is similar to existing perception-inspired compression methods. The complete system acts like a smart rendering system: It adapts dynamically to the hardware platform. Thus, it can be applied on smart-phones as well as high performance computers. The display is optimized with respect to the hardware limitations imposed. Perceptual information is extracted with the help of the early vision model. To achieve this, we introduced new methods and definitions.

Our dynamic data structure, the *TreeCut*, enables modification of an object using only two core operations: `refine` and `coarse`. The representation can be enhanced without having to determine distinct Levels Of Detail (LODs) in advance. In combination with two presented evaluation strategies – a priority-based and a distribution-based selection – multiple scenarios are covered. Both the *TreeCut* and the evaluation strategies perform well on current systems and have a linear time complexity with respect to the size of the drawn object.

The early vision model is based on a saliency calculation identifying regions of interest in a 2d map. The visual salience model is common in computer-based applications. This model does not account for higher cognitive processes and can be evaluated using only an image of the scene. We extended existing methods by incorporating 3d data allowing direct computation of saliency using object- and scene-based information. It is possible to store most of this data to avoid re-computation during run-time. Our proposed model allows to apply this information in any illumination scenario resulting in a general region of interest identification.

We connected this evaluation of the object-based saliency with the dynamic data structure and created a *Feedback System*. With help of modern Central Processing Units (CPUs) and Graphics Processing Units (GPUs), interactive frame rates are achieved. By utilizing their multi-threading architecture, stalling of individual processes, e.g. rendering, can be avoided. Because of the universal design, the *Feedback System* can be extended for various applications. As an example we present an adaptation of a physical simulation in form of a soft-body object. Soft-bodies, such as cloth, require expensive computations, and any reduction in size, e.g. simulation nodes, results in an acceleration.

In the next sections, we will discuss and evaluate our results. We will review to what extend our initial theses, stated in chapter 'Theses' on page 63, have been confirmed. We also summarize the capabilities our proposed and implemented methods with respect to the initial requirements. Finally, some notes regarding future work will be given pointing out directions of research based on our approaches.

10.1 Requirements and Theses

Previous perception-based evaluation methods in the context of 3d rendering operate mostly on computer models of early Human Visual System (HVS). In this context, the preattentive processing of perceived objects identifies regions of interest, e.g. those regions where humans are likely to look at. The visual salience model identifies objects that literally "pop-out". However, usually 2d processing of this information is performed and applied to the rendering. Only one colleague proposed an extension for 3d, but did not provide a complete 3d saliency definition.

In our first thesis, we assumed that the model of visual salience can be extended to 3d data without any restrictions and allows an identification of regions of interest on a 3d object's surface. We proposed an universal function, the Bidirectional Saliency Weight Distribution Function (BSWDF), to represent the visual salience model in a 3d context [SK11]. A mapping was defined, so that existing extractors of preattentive features can be adapted to their counterparts in 3d space. The customization of the function allows further extensions in form of different feature extractors as well as illumination models. The BSWDF enables to evaluate the object's saliency using local- and scene-based information and results in an identification of important areas.

The computation of visual salience information is not expensive, and it has been shown by others that the calculation can be performed interactively when using modern GPUs [Xu+09]. However, in real-time applications, precomputations are favored. We stated in our second thesis that object-specific features, such as surface colors and curvature, are independent of the illumination applied to the object. When deriving the definition of the BSWDF, this has been validated. The object-specific features can be stored within a lookup table to avoid costs for computing these during run-time. This computation is traded with a lookup of the associated data. The saliency is calculated during rendering by extracting object-specific features from the lookup table and combining them with an illumination model. This results in an object-based visual salience information, which can be used for evaluation and alteration of the 3d object.

Our third thesis stated that saliency information is useful when applying a LOD method to an object. This means, a reduction in size yields higher quality if a saliency-enhanced method is used. The *TreeCut*, as a *Dynamic Data Structure*, is able to utilize the saliency information, and it modifies the object representation based on predefined restrictions, such as size or rendering time [SK10]. As an prioritization of areas is available – due to the BSWDF – a perception-based reduction is performed. In user tests we have shown the validity of our third thesis and achieved higher compression rates for

saliency-enhanced methods in comparison to purely geometric-based reduction methods. Generally speaking, we are able to compress the representation of an object even further – down to 60% of the already reduced version – without introducing new distracting errors.

In conclusion, all of our theses have been validated. We created a system that operates on visual salience values and alters a 3d object representation. Details of an object are preserved during reduction without having to predefine LODs. The proposed *Feedback System* allows interactive, multi-threaded evaluation and visualization – resulting in a dynamic, saliency-influenced rendering system.

10.2 *TreeCut*

To verify thesis 3, a dynamic adaptation of a 3d object, independent of its primitive type, is required. We therefore utilized multiple LODs of an object – often referred to as multi-resolution objects –, which can be generated with either hierarchical or progressive methods. The *TreeCut* [SK10] operates on these LODs and creates a cut similar to graph-cut methods. The cut is altered by two core operations: `refine` and `coarse`. As the names imply, the `refine`-operation results in a more detailed version whereas the coarse-operation reduces the detail at the current selected node[1].

As both the `refine`- and `coarse`-operation are local operations, we propose global evaluation strategies to optimize the representation of the entire object. Either primitives are prioritized based on their importance, i.e. they are `refine`d while the unimportant ones are `coarse`ned, or a general distribution is imposed and the cut is adapted accordingly. Both strategies can be evaluated in parallel threads allowing the current representation to be redrawn without stalling the system.

When applying a LOD-reduction because the representation of the object requires to much memory or the display takes too long, common LOD transition methods require two complete levels to be drawn simultaneously. This is necessary because the direct exchange of LODs introduces artifacts that are detected easily by a human spectator. Thus, an interpolation between these two levels is needed until the substituted version can be omitted safely. However, this imposes restrictions as more primitives need to be rendered until the blending has been completed. It is also required to define distinct LODs in advance. With the evaluation strategies and a limitation, e.g. size, an object's representation given by a *TreeCut* is not only reduced to the accurate size, but also the transitions are performed in a local manner. The overhead is smaller because only local parts need to be interpolated. The *TreeCut* defines a continuous and dynamic LOD. This dynamic LOD will provide more detail as long as the system's capabilities are not exceeded while it reduces detail to retain interactivity if resources, such as battery power, need to be saved.

Also other applications for the *TreeCut* were presented. For example, a stippling rendering was achieved using the distribution-based strategy. In this case the different sampling densities in dependency of the tree level's depth are stored within a multi-resolution object. The evaluation strategy derives the distribution based on the local cut-node and current illumination scenario. It determines,

[1] We use the notions of trees or graphs, despite the fact that it can also be applied to progressive representations.

which operation needs to be performed to adapt the cut. As opposed to most existing stippling methods, ours offers frame coherency inherently. Additionally, several parameters can be changed at runtime to influence the final appearance.

The *TreeCut* can be applied for other display methods as well. We presented a highlight-based approach where the representation of a notification icon is adapted by the *TreeCut* [SSK11]. The *TreeCut* is evolved to match an external priority given, for example, by the remaining time until a task needs to be executed. As a result, the icon is altered to represent the increasing urgency of this task. We applied this approach to correlate the cognitive load of a user to the visual salience of the icon. User tests have validated that a more visual salient representation is required when the experimentee performs a mental or visual demanding task. These results build the scientific base to design a smart notification system.

10.3 Visual Salience Computation and Feature Lookup Table

To allow efficient and general computation of visual salience information within a 3d rendering framework, an universal representation, the BSWDF, is proposed [SK11]. It models saliency using a specially designed illumination and surface model. This surface model extracts preattentive features, i.e. features that are also processed by the HVS. By applying a special illumination model, these extracted features are combined and related to the current scene. This results in an identification of regions of interest similar to a saliency map.

Our proposed BSWDF represents the distribution of saliency values on the surface of an individual object. Unlike existing computation models, here 3d data is utilized and processed. This avoids the need to approximate information that is readily available within the 3d scene. Based on thesis 1, we convert the existing features to their 3d equivalents to derive a model for calculating areas of interest. Furthermore, visual salience values can be directly applied within other applications, for example, in the presented priority strategy of the *TreeCut*. As it is possible due to the general definition of the BSWDF to calculate saliency for 2d images, existing perception-based methods, which apply raytracing or global illumination algorithms, can use this model, too.

The BSWDF is a bidirectional representation, which provides functional description of a distribution based on different input parameters, similar to illumination models used in computer graphics. The calculation of the BSWDF operates with surface and scene features, for example, a distance modelling for the different stages of a center-surround inhibition. The extraction of features combines rendering and image processing. After processing, these features are weighted by the illumination model to correctly account for altered colors. Their accumulated weight results in the final saliency values. The possibility to individually weight these features need to be preserved as their weight can vary with the user situation.

It is notable that the individual surface features are independent of the illumination even when the emitted color of the surface is altered. In the BSWDF the illumination model accounts for such

changes. This independence of the light source allows to store object-based features in a lookup table (refer to thesis 2). These features are preprocessed and provide a fast access during rendering in the proposed *Feedback System*. As we wanted to design a real-time application, any online calculation should be deferred or avoided. With the lookup table, only a few instructions are required to extract the data and derive a saliency map. We proposed several sampling schemes to allow complete representation of the object features with a varying degree of detail. Once the data has been stored, a fixed space on disk is consumed, and the quality of the features is independent of the object's primitive count.

10.4 The Feedback System

The combination of the *TreeCut* and the BSWDF leads to a highly dynamic system and allows the creation of a feedback loop. The proposed *Feedback System* accounts for correct extraction and application of visual salience information by the *TreeCut*'s evaluation strategies. The 3d data given by the *TreeCut* representation is sent to the graphics card and a priority for each primitive is calculated by the BSWDF. The results are transferred to the *TreeCut*-evaluation, which adapts the current *TreeCut* representation. Once an iteration of the evaluation is completed, a new *TreeCut* representation is available. This results in new visual information, and new priorities will be assigned to each primitive. This closes the loop and creates a feedback.

We utilized both BSWDF and the lookup of the feature maps to achieve real-time updates of an object. Based on the current view, the *TreeCut* nodes are mapped to the feature map, and with the help of the BSWDF the visual salience data is computed. This mapping is enhanced by applying built-in interpolation methods of rendering subsystems, and due to the design no adaptation of the calculation is required when doing so. After this implicit mapping, each primitive contains a priority value. With the "transfer feedback" capabilities of modern graphics cards, we copy the data from GPU to CPU memory.

The *Feedback System* is self-optimizing, and this often requires additional effort to assure both a stable and optimal configuration. This restriction arises because only local information is inspected. Thus some type of control needs to be included. In the presented system we use a threshold to assure a minimal increase in detail during each iteration of the evaluation strategies. When the object has moved or the scene has been changed, new priority values are present, and the *Feedback System* optimizes the display. This preserves *TreeCut*'s ability to adapt to changes in the scene while the *Feedback System* is able to reach a stable configuration in static scenes.

An important aspect is that the evaluation is separated from the rendering process and the previous LOD-version is shown as long as the evaluation is still updating the representation. This allows to defer or even omit calculations to save processing time and to exploit multi-threading capabilities of modern computers. The more objects are present in the complete scene the more is gained from this parallel processing approach.

We have designed the complete *Feedback System* in a Model View Controller (MVC)-pattern, which allows to extend the current system with additional capabilities. The *Model* includes the raw 3d data and the assigned representations, such as the *TreeCut*. The *View* is an abstract representation for the rendering subsystem, e.g. DirectX^{TM}or a raytracer. The *Controller* manages the current state of the system and issues appropriate commands to the *Model* to update or alter the visual outcome. We added a *Feedback Stage*, which represents the loop introduced by the evaluation strategy and BSWDF computation. Our MVC-design allows combination of smart, dynamic LOD-enhanced objects along with static representations, and thus it provides flexible configurations. For example, dynamic LOD-methods can be deactivated for a single *TreeCut* to reduce evaluation costs of the complete scene.

10.5 Perception-influenced Animation

As an example application for the proposed *Feedback System*, a soft-body simulation for materials like cloth is presented [SK12b]. This type usually requires large computational effort for updating the internal forces. The less simulation nodes are available for an update the greater the increase in performance. The *Feedback System* dynamically adds or removes simulations nodes. This increases the performance of the overall system. Similar to the *TreeCut*, various types of restrictions are possible, such as size or recalculation times.

We incorporated the *TreeCut* into the simulation structure provided by a Software Development Kit (SDK), the Bullet physics engine. This way, we circumvented the need to implement correct physics calculations and showed the universal applicability of our framework. The `refine`- and the `coarse`-operations are extended to maintain the physical properties of the simulation nodes during their application. Additionally, we enhanced the existing data structure by storing incident links for each node. The locality of the `refine`- and `coarse`-operations is preserved, and thus the *TreeCut*-evaluation is accelerated.

A specialized BSWDF is defined that directly accounts for animation features, such as change of the position of a node. Animation features here identify regions where more detail in the simulation is required. These features are extracted after processing of a simulation step and the resulting saliency values are then forwarded to the *TreeCut*-evaluation, which changes the 3d representation and the physical simulation. This connection between visual representation and physical animation allows the use of a BSWDF that simultaneously accounts for visual and physical important regions.

The *Feedback System* does not need to be changed by any means as the operations directly modify both data structures. Due to the potential change in position or other physical properties, concurrent calculations of the physics engine and the *TreeCut*-evaluation have to be restricted. The evaluation may only be performed, when representation remains unchanged, e.g. when the object is drawn.

With the saliency-influenced animation we are able to change the LOD of a simulation during runtime. Here, no predefined levels have to be created in advance as before. The detail of the simulation is adapted by applying the *TreeCut*-operations while the BSWDF performs the required selection

of nodes. Our presented approach is applicable to both mesh-based and point-based representations because the simulation is independent of the surface description.

10.6 Future Work

With our proposed system, we have proven the applicability of saliency within 3d rendering scenarios. The framework fulfills the imposed requirements, but it can be enhanced even further, and several parts can be refined to increase the understanding, the universality and applicability of perception within the context of rendering.

The *TreeCut* allows to efficiently manage a representation, but a transfer of the `refine`- and `coarse`-operations onto the graphics card will lead to a more efficient processing. Not only does the GPU provide a higher computation power, but also the transfer between the CPU and the GPU would be avoided. The bandwidth saved would then be available for other information, such as occlusion queries or progressive rendering data. To avoid data transfer between CPU and GPU, a dynamic tessellation with high data throughput could be utilized. However, further studies need to be performed whether the *TreeCut* can be combined with the new capabilities offered by the shader model 5.

The application of the `refine`- and `coarse`-operation introduces an artifact at the affected node, which is unwanted. Depending on the size of the node, the saliency calculation will become invalid as we did not take the generated artifact into account. Therefore, smooth transitions of the discrete LODs are required to circumvent the introduction of visual artifacts, i.e. new salient features. The *TreeCut* needs to be adapted to account for a gradual application of both `refine`- and `coarse`-operations. This, for example, could be achieved by introducing non-discrete states.

By design, the *TreeCut* is able to operate on both a tree and a progressive representation. However, the prototype is implemented only for a tree representation, and special adjustments may be needed for a progressive data structure. Also, a compression of the representation would reduce the memory footprint, and thus it would increase performance during upload of the rendering data. Others already proposed such out-of-core upload strategies, and the *TreeCut* should be easily extended with these capabilities.

The *TreeCut* data structure currently is only applicable to point-based representations. These are well suited for highly tessellated surfaces, but when large and flat regions are present data is wasted. Like the *Progressive Meshes* approach, the *TreeCut* could be extended to mesh-based representations, too. This would broaden the range of applications. However, we aim for the use of a hybrid representation, like the POP system [CN01]. This would allow to combine the advantages of both representation types. Once the *TreeCut*-operations have been mapped to the new representation, an implementation is straightforward.

Much time is spent evaluating the *TreeCut*. Our proposed parallel implementation avoids stalling of the rendering, and only the transfer is an issue. Tests showed that a partial sorting increases performance massively, but still is too slow for large objects because sorting needs to be performed each

iteration. Therefore, more efficient approaches are required including the reuse of previously calculated data. This should be possible as between two consecutive iterations major changes are not to be expected. Currently, the data throughput of the priority strategy is rather low, and a higher rate would be appreciated, especially for large objects.

The *TreeCut* introduces an overhead due to its extended capabilities. In real-time applications, such as games, any overhead is unwanted. Additionally, the dynamic generation of LODs in fixed scenarios is usually not necessary. Thus, the question remains whether a full dynamic representation is advantageous in a real-time application. In any case, the *TreeCut* is still usable to guide a designer if creating fixed LODs.

The BSWDF needs be enhanced with multiple illumination models, so it is able to account for colored light sources. As color information suppresses some of the surface features, a relative metric would be suitable because only difference values are stored for each feature. We assume that there exists a mathematical model for a difference illumination. One has to be defined for every illumination model, but this is expected to be straightforward. Then, the influence of a light source is reflected correctly within the saliency calculation, and more general scenarios could be covered by the BSWDF.

The lookup table may be enhanced by compression algorithms to reduce memory requirements. The smaller the data the faster the upload to the graphics card, and more bandwidth is saved for other operations. Therefore, it needs be evaluated what type of data, such as precision, range, etc., is required and needs to be stored. Also, the sampling schemes presented are empirical, and the question arises how dense an object needs to be sampled to allow correct reconstruction of surface features. Suitable sampling schemes and compression methods could be verified in user tests.

The most gain of processing speed is achieved if the features are stored directly in the 3d data. We showed that the surface features are independent of the light source, but instead depend on the current view. A representation that encodes necessary information directly within the primitives avoids costly extraction and accelerates computation. For example, we approximated the surface curvature, but this could be precalculated and stored for each primitive instead. The degree of change regarding the view is an interesting research field. If it would be possible to model this change of the surface features, the necessary information could be preprocessed and stored as well.

Any recalculation generates overhead, especially for static views of an object. The calculated data could be stored and re-accessed if evaluating the current representation. If a scene is not changing up to a certain degree the final saliency values should not differ, and a cache strategy could be applied. This is similar to frame coherency and would increase the performance of the *Feedback System* as data transfer and recalculations could be avoided.

The extraction of the features in our prototype require both the CPU and GPU. The normalization operator in the current version can only be performed on the CPU. But a full GPU implementation is possible and has been proposed by others [Xu+09]. It allows interactive processing of the current scene. In combination with the precalculation of data, the accuracy of the results could be increased. Thus, no interpolation would be necessary, which is currently performed by the lookup table ap-

proach. It needs to be verified how often recalculations are required to reconstruct the surface features correctly.

The proposed BSWDF is applied only to single objects. Unlike other methods, the saliency of a complete scene then needs to be combined for all visible objects. It needs to be investigated, which type of evaluation, e.g. object- or scene-bound, is required to result in an ideal identification of regions of interest and to derive the highest quality representation. If a scene-bound version is found to be more ideal, the adaptation of an individual object could then be triggered from one global evaluation strategy while in an object-bound case each *TreeCut* would have its own evaluation process assigned.

The *Feedback System* has the drawback of requiring a threshold to establish a stable configuration. We intend to publish these results despite the fact that the the threshold approach may stop at an non-optimal state. Updates to the representation are not performed until an external change is invoked. To reduce the impact of the threshold either the data structure must be adapted to correctly model the ideal configuration or some type of memory needs to be included. Repeated changes, e.g. a `refine`-operation followed to an `coarse`-operation of the same node, needs to be avoided.

To adapt the representation with a dynamic and continuous LOD, some kind of restriction has to be imposed, which allows to optimize an object. In LOD systems, usually the size is chosen, but also rendering times may serve as a restriction. When aiming at a universal approach to render objects on various platforms, e.g. smart-phones and high performance clusters, the definition of the restrictions need to be refined. A capability representation of the current system could include rendering power, available memory, but also factors like the screen size. This is important because it limits the maximal quality a rendering can achieve and influences the saliency calculation as well. Thus, such a capability representation would support the decision making performed by the *Feedback System*.

The evaluation strategies of a *TreeCut* can optimize the representation and account for external information, in our case a prioritization of primitives. This converts the *TreeCut* with the *Feedback System* into an agent with a self-optimization property. By further refining the agent definition, multi-agent system research results should be included. The self-x properties could be enhanced including attributes like self-healing, e.g. if the representation has some type of defect. Also a Behavior-Desires-Intentions approach could be applied: The *TreeCut* would display itself with certain attributes, such as minimal quality while being restrained by the complete system. Then, the *TreeCut* would select the ideal operation to apply.

The animation presented is an example how to incorporate the *TreeCut* and the *Feedback System* to an external, existing data structure. These results have not yet been published, but have been accepted in form of a submission. To increase the quality of the results, some further steps are planned. Currently, only the operations have been validated, but the visual quality needs to be assured, too. Some user tests will be performed, and the definition refined accordingly to enable the system to retain the quality while reducing the detail in unimportant regions of the simulation.

Also, the maintenance of multiple cuts, e.g. one for the simulation and another for the rendering, could increase the overall quality. However, we state that only one underlying data structure is neces-

sary, which differs from current approaches. If a GPU-based simulation is intended, this avoids space consumption in the rather small graphics card memory. If two cuts are maintained, an update of the positions needs to be propagated from the simulation nodes to the drawn surface elements (surfels). This, of course, needs to be faster than a direct cut simulation to be reasonable. Also, it requires a mapping of surface features and the resulting saliency values onto the simulation nodes, e.g. we need to identify if a `refine`/`coarse`-operation is required on a simulation node when we calculate the saliency for the displayed surfels.

The performed user tests for rating of the rendering results only used static images. This rating, however, is assumed to be same when using animations or the dynamic *Feedback System*. Yet, this has not been verified and thus more user tests are required to evaluate the system in more detail. This, on the one hand, is only a question of performing the tests and does not affect the proposed system. On the other hand, the results may influence the computations performed and force a refinement of the definitions, such as the interpolation of nodes.

Bottom Line

We presented a step towards the incorporation of saliency into real-time rendering. Existing approaches utilized either image-based methods or these only partially included the complete saliency definition. We completed and defined a general representation of saliency for 3d objects. In combination with the *TreeCut* data structure, we are able to modify the representation of a single object instead of the complete scene. However, when solving problems by introducing new methods, new ideas and problems appeared. But the end of this study will be the beginning of future researches on this topic. So, with enthusiasm and scientific curiosity, let us restart leaded by the famous words:

> To boldly go where no man has gone before. (Star Trek Enterprise)

Bibliography

[AA03] Anders Adamson and Marc Alexa. “Ray Tracing Point Set Surfaces”. In: *Shape modeling international 2003*. Ed. by Myung-Soo Kim. Los Alamitos: IEEE Computer Society, 2003, 272–ff. ISBN: 0-7695-1909-1. URL: http://portal.acm.org/citation.cfm?id=829510.830322 (visited on 11/19/2009).

[Ada+07] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. “Adaptively sampled particle fluids”. In: *ACM Transactions on Graphics* 26.3 (2007), 48:1–48:8. ISSN: 0730-0301. DOI: 10.1145/1275808.1276437. URL: http://doi.acm.org/10.1145/1275808.1276437 (visited on 12/16/2011).

[Ale+03] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. “Computing and Rendering Point Set Surfaces”. In: *IEEE Transactions on Visualization and Computer Graphics* 9.1 (2003), pp. 3–15. ISSN: 1077-2626. URL: http://dl.acm.org/citation.cfm?id=614289.614541 (visited on 12/16/2011).

[Are+05] Vutipong Areekul, Ukrit Watchareeruetai, Kittiwat Suppasrawasuseth, and Saward Tantaratana. “Separable Gabor Filter Realization for Fast Fingerprint Enhancement”. In: *2005 International Conference on Image Processing (ICIP)*. Piscataway: IEEE Operations Center, 2005, pp. 253–256. ISBN: 0-7803-9134-9. DOI: 10.1109/ICIP.2005.1530376. URL: http://dx.doi.org/10.1109/ICIP.2005.1530376 (visited on 12/16/2011).

[BA08] Tamy Boubekeur and Marc Alexa. “Phong Tesselation”. In: *ACM Transactions on Graphics* 27.5 (2008), 141:1–5. ISSN: 0730-0301. URL: http://doi.acm.org/10.1145/1457515.1409094 (visited on 12/16/2011).

[BK04] Jacob Beaudoin and John Keyser. “Simulation levels of detail for plant motion”. In: *Computer animation 2004*. Ed. by R. Boulic and D. K. Pai. Aire-la-Ville: Eurographics Association, 2004, pp. 297–304. ISBN: 3-905673-14-2. DOI: 10.1145/1028523.1028563. URL: http://dx.doi.org/10.1145/1028523.1028563 (visited on 12/16/2011).

[Bon11] Xavier Bonaventura. “GPU Pro 2: Terrain and Ocean Rendering with Hardware Tessellation”. In: *GPU Pro2*. Ed. by Wolfgang F. Engel. Natick: AK Peters, 2011. ISBN: 1-5688-1718-5. URL: http://gilab.udg.edu/publ/GraphicsImagingPublications (visited on 12/16/2011).

[Bot+05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif P. Kobbelt. “High-Quality Surface Splatting on Today’s GPUs”. In: *Eurographics Symposium on Point-Based Graphics*. Eurographics Association, 2005, pp. 17–24. URL: http://diglib.eg.org/EG/DL/WS/SPBG/SPBG05/017-024.pdf (visited on 12/13/2011).

[BSK04] Mario Botsch, Michael Spernat, and Leif P. Kobbelt. “Phong Splatting”. In: *Symposium on Point-Based Graphics 2004*. Ed. by Markus Gross, Hanspeter Pfister, Marc Alex, and Szymon Rusinkiewicz. Aire-la-Ville: Eurographics Assoc., 2004, pp. 25–32. ISBN: 3-905673-09-6. URL: http://diglib.eg.org/EG/DL/WS/SPBG/SPBG04/025-032.pdf (visited on 12/13/2011).

[BSW08] Peter Barth, Ulrich Schwanecke, and Friederike Wild. “Interactive high definition 3D face rendering on common mobile devices”. In: *Mensch & Computer*. Ed. by Michael Herczeg and Martin Christof Kindsmüller. Oldenbourg Verlag, 2008, pp. 197–206. URL: http://dblp.uni-trier.de/db/conf/mc/mc2008.html#BarthSW08 (visited on 12/16/2011).

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. “Efficient High Quality Rendering of Point Sampled Geometry”. In: *13th Eurographics Workshop on Rendering*. Ed. by Simon Gibson and Paul E. Debevec. New York: Association for Computing Machinery, 2002, pp. 53–64. ISBN: 1-58113-534-3. URL: http://dl.acm.org/citation.cfm?id=581896.581904 (visited on 12/13/2011).

[Cal93] Paul B. Callahan. “Optimal Parallel All-Nearest-Neighbors Using the Well-Separated Pair Decomposition”. In: *Proceedings / 34th Annual Symposium on Foundations of Computer Science*. Los Alamitos: IEEE Computer Soc. Pr., 1993, pp. 332–340. ISBN: 0-8186-4370-6. DOI: 10.1109/SFCS.1993.366854. URL: http://dl.acm.org/citation.cfm?id=1398517.1398918 (visited on 12/16/2011).

[CB05] Irene Cheng and Pierre Boulanger. “Automatic Selection of Level-of-detail based on Just-Noticeable-Differenc (JND)”. In: *Proceedings of ACM SIGGRAPH 2005*. Ed. by Markus Gross. New York: ACM, 2005, p. 102. URL: http://doi.acm.org/10.1145/1186954.1187070 (visited on 12/16/2011).

[CCW03] Kirsten Cater, Alan Chalmers, and Gregory J. Ward. “Detail to Attention: Exploiting Visual Tasks for Selective Rendering”. In: *Eurographics Symposium on Rendering*. Ed. by Frank Suykens, Philip Dutré, Per H. Christensen, and Daniel Cohen-Or. New York: ACM, 2003, pp. 270–280. ISBN: 3-905673-03-7. URL: http://portal.acm.org/citation.cfm?id=882404.882443 (visited on 11/19/2009).

[CF11] Rhadamés Carmona and Bernd Froehlich. “Error-controlled real-time cut updates for multi-resolution volume rendering”. In: *Computers & Graphics* 35.4 (2011), pp. 934–944. ISSN: 0097-8493. URL: http://dx.doi.org/10.1016/j.cag.2011.01.007 (visited on 12/16/2011).

[CK10] Michael Connor and Piyush Kumar. “Fast Construction of k-Nearest Neighbor Graphs for Point Clouds”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.4 (2010), pp. 599–608. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.9. URL: http://dx.doi.org/10.1109/TVCG.2010.9 (visited on 12/16/2011).

[CN01] Boaquan Chen and Minh X. Nguyen. “POP: A Hybrid Point and Polygon Rendering System for Large Data”. In: *Visualization 2001*. Ed. by Thomas Ertl, Ken Joy, and Amitabh Varshney. Piscataway: IEEE Computer Society, 2001, pp. 45–52. ISBN: 0-7803-7200-X. URL: http://portal.acm.org/citation.cfm?id=601676 (visited on 12/16/2011).

[Coh+95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. “I-COLLIDE: an interactive and exact collision detection system for large-scale environments”. In: *Proceedings of the 1995 Symposium on Interactive 3D Graphics*. New York: ACM, 1995, 189–ff. ISBN: 978-0-89791-736-0. DOI: 10.1145/199404.199437. URL: http://doi.acm.org/10.1145/199404.199437 (visited on 12/16/2011).

[Coo84] Robert L. Cook. “Shade trees”. In: *SIGGRAPH Comput. Graph.* 18.3 (1984), pp. 223–231. ISSN: 0097-8930. DOI: 10.1145/964965.808602. URL: http://doi.acm.org/10.1145/964965.808602 (visited on 12/16/2011).

[Cor74] Tom N. Cornsweet. *Visual perception*. 3. printing. New York and NY: Acad. Pr., 1974. ISBN: 0121897508.

[Cra+09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. “GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering”. In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. Ed. by Eric Haines, Daniel G. Aliaga, and Stephen N. Spencer. New York: ACM, 2009, pp. 15–22. ISBN: 978-1-60558-429-4. DOI: 10.1145/1507149.1507152. URL: http://doi.acm.org/10.1145/1507149.1507152 (visited on 12/16/2011).

[CV95] James E. Cutting and Peter M. Vishton. “Perceiving Layout and Knowing Distances: The Integration, Relative Potency, and Contextual Use of Different Information about Depth”. In: *Perception of space and motion*. Ed. by William Epstein and Sheena J. Rogers. Handbook of perception and cognition (2nd ed.) San Diego: Academic Press, 1995, pp. 69–117. ISBN: 978-0-12-240530-3. DOI: 10.1016/B978-012240530-3/50005-5. URL: http://dx.doi.org/10.1016/B978-012240530-3/50005-5 (visited on 12/13/2011).

[Dal93] Scott Daly. “The Visible Differences Predictor: An Algorithm for the Assessment of Image Fidelity”. In: *Digital images and human vision*. Ed. by Andrew B. Watson. Cambridge: MIT Press, 1993, pp. 179–206. ISBN: 0-262-23171-9. URL: http://dl.acm.org/citation.cfm?id=197765.197783 (visited on 12/13/2011).

[Dal98] Scott J. Daly. “Engineering observations from spatiovelocity and spatiotemporal visual models”. In: *Proceedings of SPIE 3299* 3299.1 (1998), pp. 180–193. ISSN: 0361-0748. URL: http://direct.bl.uk/bld/PlaceOrder.do?UIN=048353708&ETOC=RN&from=searchengine (visited on 12/16/2011).

[Dre+07] George Drettakis, Nicolas Bonneel, Carsten Dachsbacher, Sylvain Lefebvre, Michael Schwarz, and Isabelle Viaud-Delmon. "An Interactive Perceptual Rendering Pipeline using Contrast and Spatial Masking". In: *Rendering techniques 2007*. Ed. by Nicolas Holzschuh, Jan Kautz, and Sumanta Pattanaik. Aire-la-Ville: Eurographics Asso., 2007, pp. 297–308. ISBN: 978-3-905673-52-4. URL: http://diglib.eg.org/EG/DL/WS/EGWR/EGSR07/297-308.pdf (visited on 12/16/2011).

[DVS03] Carsten Dachsbacher, Christian Vogelsang, and Marc Stamminger. "Sequential point trees". In: *ACM Transactions on Graphics* 22.3 (2003), pp. 657–662. ISSN: 0730-0301. URL: http://portal.acm.org/citation.cfm?id=1201775.882321 (visited on 11/19/2009).

[FCH08] Jiri Filip, Michael J. Chantler, and Michal Haindl. "On Optimal Resampling of View and Illumination Dependent Textures". In: *Proceedings APGV 2008*. Ed. by Sarah Creem-Regehr, K. Myszkowski, Bobby Bodenheimer, Betty J. Mohler, Bernhard Riecke, and Stephen N. Spencer. New York: ACM Press, 2008, pp. 131–134. ISBN: 978-1-59593-981-4. URL: http://doi.acm.org/10.1145/1394281.1394305 (visited on 12/16/2011).

[Fer04] Randima Fernando. *GPU gems: Programming techniques, tips, and tricks for real-time graphics*. 5. printing. Vol. 1. Boston: Addison-Wesley, 2004. ISBN: 0-321-22832-4.

[FKF06] Randima Fernando, Mark J. Kilgard, and Fernando-Kilgard. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. 7. printing. Boston: Addison-Wesley, 2006. ISBN: 0-321-19496-9.

[FP04] Jean Phillipe Farrugia and Bernard Peroche. "A Progressive Rendering Algorithm Using an Adaptive Perceptually Based Image Metric". In: *Computer Graphics forum*. Vol. 23/3. Blackwell Publishing, Inc, 2004, pp. 605–614. URL: http://www3.interscience.wiley.com/journal/120843879/abstract?CRETRY=1&SRETRY=0 (visited on 11/13/2009).

[FS10] John P. Frisby and James V. Stone. *Seeing: The computational approach to biological vision*. 2. ed.. Cambridge: MIT Press, 2010. ISBN: 978-0-262-51427-9.

[FSG09] Miquel Feixas, Mateu Sbert, and Francisco Gonzalez. "A unified information-theoretic framework for viewpoint selection and mesh saliency". In: *ACM Transactions on Applied Perception* 6.1 (2009), 1:1–23. ISSN: 1544-3558. DOI: 10.1145/1462055.1462056. URL: http://portal.acm.org/citation.cfm?id=1462056 (visited on 11/13/2009).

[GDO08] Marcos Garcia, John Dingliana, and Carol O'Sullivan. "Perceptual Evaluation of Cartoon Physics: Accuracy, Attention, Appeal". In: *Proceedings APGV 2008*. Ed. by Sarah Creem-Regehr, K. Myszkowski, Bobby Bodenheimer, Betty J. Mohler, Bernhard Riecke, and Stephen N. Spencer. New York: ACM Press, 2008, pp. 107–114. ISBN: 978-1-59593-

981-4. DOI: 10.1145/1394281.1394301. URL: http://doi.acm.org/10.1145/1394281.1394301 (visited on 12/16/2011).

[GG07] Gaël Guennebaud and Markus Gross. "Algebraic point set surfaces". In: *ACM Transactions on Graphics* 26.3 (2007), 23:1–23:9. ISSN: 0730-0301. URL: http://portal.acm.org/citation.cfm?id=1275808.1276406 (visited on 11/20/2009).

[GGH02] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. "Geometry images". In: *SIGGRAPH '02*. Vol. 29. International Conference on Computer Graphics and Interactive Techniques. New York: ACM, 2002, pp. 355–361. URL: http://portal.acm.org/citation.cfm?id=566570.566589 (visited on 11/19/2009).

[GH97] Michael Garland and Paul S. Heckbert. "Surface Simplification using quadric error metrics". In: *The art and interdisciplinary programs of SIGGRAPH 97*. Computer graphics annual conference series. New York: ACM SIGGRAPH, 1997, pp. 209–216. ISBN: 0-89791-896-7. DOI: 10.1145/258734.258849. URL: http://dx.doi.org/10.1145/258734.258849 (visited on 12/13/2011).

[Gla95] Andrew S. Glassner. *Principles of digital image synthesis*. San Francisco: Morgan Kaufmann, 1995. ISBN: 978-1558602762.

[Gos96] Rich Gossweiler. "Perception-based time critical rendering". MA thesis. Charlottesville: University Of Virginia, 1996.

[GP07] Markus Gross and Hanspeter Pfister. *Point-based graphics*. The Morgan Kaufmann series in computer graphics. Amsterdam: Morgan Kaufmann/Elsevier, 2007. ISBN: 978-0-12-370604-1.

[Gre+03] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. "Real-time procedural generation of 'pseudo infinite' cities". In: *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. Ed. by Matt Adcock and Stephen N. Spencer. New York: Association for Computing Machinery, 2003, pp. 87–95. ISBN: 1-58113-578-5. URL: http://dl.acm.org/citation.cfm?id=604490 (visited on 12/13/2011).

[Gri+11] Wesley Griffin, Yu Wang, David Berrios, and Marc Olano. "GPU curvature estimation on deformable meshes". In: *Symposium on Interactive 3D Graphics and Games*. I3D '11. New York: ACM, 2011, pp. 159–166. ISBN: 978-1-4503-0565-5. DOI: 10.1145/1944745.1944772.

[Gus+00] Igor Guskov, Kiril Vidimce, Wim Sweldens, and Peter Schroeder. "Normal meshes". In: *SIGGRAPH 2000 conference proceedings*. Ed. by Kurt Akeley. New York: ACM Press, 2000, pp. 95–102. ISBN: 1-58113-208-5. URL: http://portal.acm.org/citation.cfm?id=344831 (visited on 11/19/2009).

[GW07] Markus Giegl and Michael Wimmer. “Unpopping: Solving the Image-Space Blend Problem for Smooth Discrete LOD Transitions”. In: *Computer Graphics Forum* 26.1 (2007), pp. 46–49. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2007.00943.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2007.00943.x (visited on 12/16/2011).

[Hab+01] Joerg Haber, Karol Myszkowski, Hitoshi Yamauchi, and Hans-Peter Seidel. “Perceptually Guided Corrective Splatting”. In: *Computer Graphics Forum* 20.3 (2001), pp. 142–153. ISSN: 1467-8659. URL: http://www.ingentaconnect.com/content/bpl/cgf/2001/00000020/00000003/art00507 (visited on 11/19/2009).

[HDK01] Stefan Hiller, Oliver Deussen, and Alexander Keller. “Tiled Blue Noise Samples”. In: *Vision, modeling, and visualization 2001*. Ed. by Thomas Ertl. Berlin: AKA, 2001, pp. 265–272. ISBN: 3-89838-028-9. URL: http://dl.acm.org/citation.cfm?id=647260.718507 (visited on 12/16/2011).

[Hop+92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. “Surface Reconstruction from Unorganized Points”. In: *SIGGRAPH’92 Conference proceedings*. Ed. by Edwin E. Catmull. New York: ACM Press, 1992, pp. 71–78. ISBN: 0-89791-479-1. DOI: 10.1145/133994.134011. URL: http://doi.acm.org/10.1145/133994.134011 (visited on 12/13/2011).

[Hop96] Hugues Hoppe. “Progressive Meshes”. In: *SIGGRAPH 96 conference proceedings*. Ed. by Holly Rushmeier. Vol. 1996. New York: ACM, 1996, pp. 99–108. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237216. URL: http://doi.acm.org/10.1145/237170.237216 (visited on 12/16/2011).

[IKN98] Laurent Itti, Christof Koch, and Ernst Niebur. “A model of saliency-based visual attention for rapid scene analysis”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.11 (1998), pp. 1254–1259. ISSN: 0162-8828. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?isnumber=15773&arnumber=730558&punumber=34 (visited on 12/16/2011).

[Itt05] Laurent Itti. “Models of bottom-up attention and saliency”. In: *Neurobiology of attention*. Ed. by Laurent Itti, Geraint Rees, and John K. Tsotsos. Amsterdam: Elsevier Academic Press, 2005, pp. 576–582. ISBN: 978-0123757319. URL: http://ilab.usc.edu/publications/doc/Itti05noa.pdf (visited on 12/13/2011).

[Kel79] D. H. Kelly. “Motion and vision. II. Stabilized spatio-temporal threshold surface”. In: *Journal of the Optical Society of America* 69.10 (1979), pp. 1340–1349. ISSN: 1084-7529. URL: http://www.opticsinfobase.org/abstract.cfm?URI=josa-69-10-1340 (visited on 12/16/2011).

[KU85] Christof Koch and Shimon Ullman. “Shifts in selective visual attention: towards the underlying neural circuitry”. In: *Human Neurobiology* 4.4 (1985), pp. 219–227. ISSN: 0721-9075. URL: http://www.ncbi.nlm.nih.gov/pubmed/3836989 (visited on 12/13/2011).

[KV03] Aravind Kalaiah and Amitabh Varshney. “Statistical point geometry”. In: *Symposium on Geometry Processing*. Ed. by Stephen N. Spencer, Leif Kobbelt, Peter Schröder, and Hugues Hoppe. Aire-la-Ville (PO Box 16 and 1288): Eurographics Association, 2003, pp. 107–115. ISBN: 1-58113-687-0. URL: http://portal.acm.org/citation.cfm?id=882370.882385 (visited on 12/16/2011).

[Lau] Andrew Lauritzen. *Deferred Rendering for Current and Future Rendering Pipelines*. URL: http://software.intel.com/en-us/articles/deferred-rendering-for-current-and-future-rendering-pipelines/ (visited on 12/15/2011).

[LC87] William E. Lorensen and Harvey E. Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. In: *Proceedings of SIGGRAPH ’87*. New York: ACM, 1987, pp. 163–169. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37422. URL: http://doi.acm.org/10.1145/37401.37422 (visited on 12/16/2011).

[LCD06] Thomas Luft, Carsten Colditz, and Oliver Deussen. “Image enhancement by unsharp masking the depth buffer”. In: *ACM Transactions on Graphics* 25.3 (2006), pp. 1206–1213. ISSN: 0730-0301. URL: http://doi.acm.org/10.1145/1141911.1142016 (visited on 12/16/2011).

[Le +06] Olivier Le Meur, Patrick Le Callet, Dominique Barba, and Dominique Thoreau. “A coherent computational approach to model the bottom–up visual attention”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.5 (2006), pp. 802–817. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2006.86. URL: http://dx.doi.org/10.1109/TPAMI.2006.86 (visited on 12/16/2011).

[LM05] Peter Lennie and J. Anthony Movshon. “Coding of color and form in the geniculostriate visual pathway (invited review)”. In: *Journal of The Optical Society of America A-optics Image Science and Vision* 22.10 (2005), pp. 2013–2033. ISSN: 1084-7529. DOI: 10.1364/JOSAA.22.002013. URL: http://dx.doi.org/10.1364/JOSAA.22.002013 (visited on 12/13/2011).

[Loo+09] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. “Approximating subdivision surfaces with Gregory patches for hardware tessellation”. In: *ACM Transactions on Graphics* 28 (2009), 151:1–151:9. ISSN: 0730-0301. DOI: 10.1145/1618452.1618497. URL: http://doi.acm.org/10.1145/1618452.1618497 (visited on 12/13/2011).

[LS08] Charles Loop and Scott Schaefer. "Approximating Catmull-Clark subdivision surfaces with bicubic patches". In: *ACM Transactions on Graphics* 27 (2008), 8:1–8:11. ISSN: 0730-0301. DOI: 10.1145/1330511.1330519. URL: http://doi.acm.org/10.1145/1330511.1330519 (visited on 12/16/2011).

[Lue03] David P. Luebke. *Level of detail for 3D graphics*. 1st ed. The Morgan Kaufmann series in computer graphics and geometric modeling. Boston: Morgan Kaufmann, 2003. ISBN: 1-55860-838-2.

[LVJ05] Chang Ha Lee, Amitabh Varshney, and David W. Jacobs. "Mesh saliency". In: *Proceedings of ACM SIGGRAPH 2005*. Ed. by Markus Gross. New York: ACM, 2005, pp. 659–666. URL: http://portal.acm.org/citation.cfm?id=1073244 (visited on 11/17/2009).

[Mar+10] Domingo Martin, Germán Arroyo, M. Victoria Luzón, and Tobias Isenberg. "Example-based stippling using a scale-dependent grayscale process". In: *Proceedings NPAR 2010*. Ed. by Morgan McGuire, John Collomosse, and Stephen N. Spencer. New York: ACM, 2010, pp. 51–61. ISBN: 978-1-4503-0125-1. DOI: 10.1145/1809939.1809946. URL: http://doi.acm.org/10.1145/1809939.1809946 (visited on 12/16/2011).

[MCA07] M. Mousa, Raphaelle Chaine, and S. Akkouche. "Frequency-Based Representation of 3D Models using Spherical Harmonics". In: *WSCG'2006 - 14-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2006*. Ed. by Vaclav Skala. Plzen: Univ. of West Bohemia, 2007, pp. 193–200. ISBN: 80-86943-03-8. URL: http://wscg.zcu.cz/wscg2006/Papers_2006/Full/A53-full.pdf (visited on 12/13/2011).

[Mil10] Ian Millington. *Game physics engine development: How to build a robust commercial-grade physics engine for your game*. 2nd ed. Amsterdam: Morgan Kaufmann Publishers, 2010. ISBN: 978-0123819765. URL: http://www.sciencedirect.com/science/book/9780123819765.

[MS74] J. L. Mannos and D. J. Sakrison. "The effects of a visual fidelity criterion on the encoding of images". In: *IEEE Transactions on Information Theory* 20.4 (1974), pp. 525–536. ISSN: 0018-9448. DOI: 10.1109/TIT.1974.1055250. URL: http://dx.doi.org/10.1109/TIT.1974.1055250 (visited on 12/13/2011).

[Mue+04] Matthias Mueller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus Gross, and Marc Alexa. "Point based animation of elastic, plastic and melting objects". In: *Computer animation 2004*. Ed. by R. Boulic and D. K. Pai. Aire-la-Ville: Eurographics Association, 2004, pp. 141–151. ISBN: 3-905673-14-2. URL: http://portal.acm.org/citation.cfm?id=1028542 (visited on 11/20/2009).

[Mun+10] Jacob Munkberg, Jon Hasselgren, Robert Toth, and Tomas Akenine-Möller. "Efficient bounding of displaced Bézier patches". In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Aire-la-Ville: Eurographics Association, 2010, pp. 153–162. URL: http://portal.acm.org/citation.cfm?id=1921479.1921503 (visited on 12/13/2011).

[Mys02] Karol Myszkowski. "Perception-based global illumination, rendering, and animation techniques". In: *Spring Conference on Computer Graphics*. 2002, pp. 13–24. URL: http://portal.acm.org/citation.cfm?id=584458.584462 (visited on 11/13/2009).

[NBZ07] Jeffrey Ng, Anil A. Bharath, and Li Zhaoping. "A survey of architecture and function of the primary visual cortex (V1)". In: *EURASIP Journal on Applied Signal Processing* 2007.1 (2007), pp. 124–124. ISSN: 1110-8657. URL: http://portal.acm.org/citation.cfm?id=1289085 (visited on 11/13/2009).

[Ngu08] Hubert Nguyen. *GPU gems 3*. Upper Saddle River: Addison-Wesley, 2008. ISBN: 0-321-51526-9. URL: http://www.loc.gov/catdir/toc/ecip0720/2007023985.html.

[Not00] Hans-Christoph Nothdurft. "Salience from feature contrast: variations with texture density". In: *Vision Research* 40.23 (2000), pp. 3181–3200. ISSN: 0042-6989. DOI: 10.1016/S0042-6989(00)00168-1. URL: http://www.sciencedirect.com/science/article/pii/S0042698900001681 (visited on 12/16/2011).

[NY06] T. Newman and Hong Yi. "A survey of the marching cubes algorithm". In: *Computers & Graphics* 30.5 (2006), pp. 854–879. ISSN: 0097-8493. URL: http://www.sciencedirect.com/science/article/pii/S0097849306001336 (visited on 12/13/2011).

[OO11] Michael Oberguggenberger and Alexander Ostermann. *Analysis for Computer Scientists: Foundations Methods and Algorithms*. London: Springer-Verlag London Limited, 2011. ISBN: 978-0-85729-445-6.

[OSu05] Carol O'Sullivan. "Collisions and Attention". In: *ACM Transactions on Applied Perception* 2.3 (2005), pp. 309–321. ISSN: 1544-3558. URL: http://portal.acm.org/citation.cfm?id=1077399.1077407 (visited on 11/20/2009).

[Pal11] Sven Pallus. "Point-Based Animation". MA thesis. Frankfurt: Goethe Universität, 2011.

[Par08] Rick Parent. *Computer animation: Algorithms and techniques*. 2. ed. The Morgan Kaufmann series in computer graphics. Amsterdam: Elsevier Morgan Kaufmann, 2008. ISBN: 9780125320009. URL: http://www.gbv.de/dms/ilmenau/toc/543356094.PDF.

[Pat+98] Sumanta N. Pattanaik, James A. Ferwerda, Mark D. Fairchild, and Donald P. Greenberg. "A multiscale model of adaptation and spatial vision for realistic image display". In: *SIGGRAPH 98 conference proceedings*. Ed. by Michael Cohen. New York: ACM Press,

1998, pp. 287–298. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280922. URL: http://doi.acm.org/10.1145/280814.280922 (visited on 12/16/2011).

[PF05] M. Pharr and R. Fernando. *GPU gems 2: Programming techniques for high- performance graphics and general-purpose computation*. Upper Saddle River: Addison-Wesley, 2005. ISBN: 978-0-321-33559-3. URL: http://www.loc.gov/catdir/toc/ecip055/2004030181.html.

[Pfi+00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. "Surfels: surface elements as rendering primitives". In: *SIGGRAPH 2000 conference proceedings*. Ed. by Kurt Akeley. New York: ACM Press, 2000, pp. 335–342. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344936. URL: http://dx.doi.org/10.1145/344779.344936 (visited on 12/16/2011).

[PG01] Mark Pauly and Markus Gross. "Spectral processing of point-sampled geometry". In: *SIGGRAPH 2001 conference proceedings*. Ed. by Eugene Fiume. New York: ACM, 2001, pp. 379–386. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383301. URL: http://doi.acm.org/10.1145/383259.383301 (visited on 12/13/2011).

[PGK02] Mark Pauly, Markus Gross, and Leif P. Kobbelt. "Efficient simplification of point-sampled surfaces". In: *VIS2002*. Ed. by Robert Moorhead, Markus Gross, and Kenneth I. Joy. Washington: IEEE, 2002, pp. 163–170. ISBN: 0-7803-7498-3. URL: http://portal.acm.org/citation.cfm?id=602099.602123 (visited on 12/16/2011).

[PKG06] Mark Pauly, Leif P. Kobbelt, and Markus Gross. "Point-Based MultiScale Surface Representation". In: *ACM Transactions on Graphics* 25.2 (2006), pp. 177–193. ISSN: 0730-0301. DOI: 10.1145/1138450.1138451. URL: http://doi.acm.org/10.1145/1138450.1138451 (visited on 12/13/2011).

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. "Real-time relief mapping on arbitrary polygonal surfaces". In: *Proceedings I3D 2005 : ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. Ed. by David P. Luebke, Hanspeter Pfister, Anselmo Lastra, and Marc Olano. New York: ACM, 2005, pp. 155–162. ISBN: 1-59593-013-2. DOI: 10.1145/1053427.1053453. URL: http://doi.acm.org/10.1145/1053427.1053453 (visited on 12/16/2011).

[Pur+02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. "Ray Tracing on Programmable Graphics Hardware". In: *ACM Transactions on Graphics* 21.3 (2002), pp. 703–712. ISSN: 0730-0301. URL: http://doi.acm.org/10.1145/566654.566640 (visited on 12/16/2011).

[Red97] Martin Reddy. "Perceptually modulated level of detail for virtual environments". MA thesis. University of Edinburgh, 1997.

[RL00] Szymon Rusinkiewicz and Marc Levoy. “QSplat: a multiresolution point rendering system for large meshes”. In: *SIGGRAPH 2000 conference proceedings*. Ed. by Kurt Akeley. New York: ACM Press, 2000, pp. 343–352. ISBN: 1-58113-208-5. URL: http://dx.doi.org/10.1145/344779.344940 (visited on 12/16/2011).

[Rog01] David F. Rogers. *An introduction to NURBS: With historical perspective*. San Francisco: Morgan Kaufmann, 2001. ISBN: 1558606696. URL: http://www.loc.gov/catdir/description/els031/00039119.html.

[RPG99] Mahesh Ramasubramanian, Sumanta N. Pattanaik, and Donald P. Greenberg. “A perceptually based physical error metric for realistic image synthesis”. In: *SIGGRAPH 99 conference proceedings*. Ed. by Alyn Paul Rockwood. New York: ACM, 1999, pp. 73–82. ISBN: 0-201-48560-5. DOI: 10.1145/311535.311543. URL: http://dx.doi.org/10.1145/311535.311543 (visited on 12/16/2011).

[RPZ02] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. “Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering”. In: *Computer Graphics Forum* 21.3 (2002), pp. 461–470. ISSN: 1467-8659. DOI: 10.1111/1467-8659.00606. URL: http://dx.doi.org/10.1111/1467-8659.00606 (visited on 12/16/2011).

[Rus04] Szymon Rusinkiewicz. “Estimating Curvatures and Their Derivatives on Triangle Meshes”. In: *Proceedings*. Ed. by Yiannis Aloimonos. Los Alamitos and Calif.: IEEE Computer Society, 2004, pp. 486–493. ISBN: 0-7695-2223-8. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1335277 (visited on 12/13/2011).

[Sam84] Hanan Samet. “The Quadtree and Related Hierarchical Data Structures”. In: *ACM Computing Surveys* 16.2 (1984), pp. 187–260. ISSN: 0360-0300. DOI: 10.1145/356924.356930. URL: http://dl.acm.org/citation.cfm?id=356930 (visited on 12/16/2011).

[Sam94] Hanan Samet. *The design and analysis of spatial data structures*. Reading: Addison-Wesley, 1994. ISBN: 0-201-50255-0.

[Sam95] Hanan Samet. *Applications of spatial data structures: Computer graphics, image processing and GIS*. Reading: Addison-Wesley, 1995. ISBN: 978-0-201-50300-5.

[SH10] Sebastian Schäfer and Carsten Heep. “3D-Oberflächen-Rekonstruktion und plastisches Rendern aus Bildserien”. In: *Tagungsband 16. Workshop Farbbildverarbeitung Ilmenau*. Ed. by K. H. Franke and R. Nestler. 2010, pp. 193–204. ISBN: 8-3-00-032504-5. URL: http://www.gdv.informatik.uni-frankfurt.de/forschung/nibbla/PaFWS2010.pdf (visited on 12/16/2011).

[SK10] Daniel Schiffner and Detlef Krömker. “Tree-Cut: Dynamic Saliency Based Level of Detail for Point Based Rendering”. In: *Self Integrating Systems for Better Living Environments: First Workshop, Sensyble 2010*. Ed. by Ralf Dörner and Detlef Krömker. Shaker Aachen, 2010, pp. 37–43. URL: http://www.sensyble.org/lang/de/publikationen/ (visited on 12/16/2011).

[SK11] Daniel Schiffner and Detlef Krömker. “Three Dimensional Saliency Calculation Using Splatting”. In: *Sixth International Conference on Image and Graphics (ICIG), 2011*. Ed. by Nenghai Yu. Piscataway: IEEE, 2011, pp. 835–840. ISBN: 978-1-4577-1560-0. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6005958 (visited on 12/13/2011).

[SK12a] Daniel Schiffner and Detlef Krömker. “Parallel Treecut-Manipulation for Interactive Level of Detail Selection (to appear)”. In: *WSCG*. 2012,

[SK12b] Daniel Schiffner and Detlef Krömker. “Perception-influenced Animation”. In: *ARCS Workshop on Architectures for Self-Organizing PRivate IT-Spheres*. Springer, 2012, pp. 123–134.

[SMA10] Peter Shirley, Stephen Robert Marschner, and Michael Ashikhmin. *Fundamentals of computer graphics*. 3. ed., [revised]. Natick: AK Peters, 2010. ISBN: 978-1-56-881469-8.

[SS00] Andrew Stockman and Lindsay T. Sharpe. “The spectral sensitivities of the middle- and long-wavelength-sensitive cones derived from measurements in observers of known genotype”. In: *Vision Research* 40.13 (2000), pp. 1711–1737. ISSN: 0042-6989. URL: http://www.sciencedirect.com/science/article/pii/S0042698900000213 (visited on 12/16/2011).

[SSK11] Claudia Stockhausen, Daniel Schiffner, and Detlef Krömker. “Adaptation of User Interfaces based on Physiological Data”. In: *9. Berliner Werkstatt Mensch-Maschine-Systeme (BWMMS)* (2011), pp. 78–79.

[SSV05] Jagan Sankaranarayanan, Hanan Samet, and Amitabh Varshney. “A Fast k-Neighborhood Algorithm for Large Point-Clouds”. In: *Eurographics Symposium on Point-Based Graphics*. Eurographics Association, 2005, pp. 75–84. URL: http://diglib.eg.org/EG/DL/WS/SPBG/SPBG06/075-084.pdf (visited on 12/13/2011).

[SWD05] Gaurav Sharma, Wencheng Wu, and Edul N. Dalal. “The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations”. In: *Color Research & Application* 30.1 (2005), pp. 21–30. ISSN: 1520-6378. DOI: 10.1002/col.20070. URL: http://dx.doi.org/10.1002/col.20070 (visited on 12/16/2011).

[Tau95] Gabriel Taubin. “Estimating the tensor of curvature of a surface from a polyhedral approximation”. In: *Fifth International Conference on Computer Vision*. Los Alamitos: IEEE Computer Society, 1995, pp. 902–. ISBN: 0-8186-7042-8. URL: http://dl.acm.org/citation.cfm?id=839277.840020 (visited on 12/16/2011).

[TG80] Anne M. Treisman and Garry Gelade. “A feature-integration theory of attention”. In: *Cognitive Psychology* 12.1 (1980), pp. 97–136. ISSN: 0010-0285. DOI: 10.1016/0010-0285(80)90005-5. URL: http://www.sciencedirect.com/science/article/pii/0010028580900055 (visited on 12/16/2011).

[War00] Colin Ware. *Information visualization: Perception for design*. San Francisco: Morgan Kaufmann, 2000. ISBN: 1-55860-511-8.

[WCF10] D. A. Warrell, Timothy M. Cox, and John D. Firth. *Oxford textbook of medicine*. 5. ed. Oxford: Oxford Univ. Press, 2010. ISBN: 978-0199204854.

[WK06] Dirk Walther and Christof Koch. “Modeling attention to salient proto-objects”. In: *Neuronal Networks*. Vol. 19/9. Elsevier Science Ltd., 2006, pp. 1395–1407. URL: http://www.sciencedirect.com/science/article/pii/S0893608006002152 (visited on 12/16/2011).

[WP01] Alan H. Watt and Fabio Policarpo. *3D games*. ACM SIGGRAPH series. Harlow: Addison-Wesley, 2001.

[WS06] Michael Wimmer and Claus Scheiblauer. “Instant Points”. In: *Proceedings Symposium on Point-Based Graphics 2006*. Eurographics Association, 2006, pp. 129–136. ISBN: 3-90567-332-0. URL: http://www.cg.tuwien.ac.at/research/publications/2006/WIMMER-2006-IP/ (visited on 12/16/2011).

[WZK05] Jianhua Wu, Zhou Zhang, and Leif P. Kobbelt. “Progressive Splatting”. In: *Eurographics Symposium on Point-Based Graphics*. Eurographics Association, 2005, pp. 25–32. DOI: 10.2312/SPBG/SPBG05/025-032. URL: http://diglib.eg.org/EG/DL/WS/SPBG/SPBG05/025-032.pdf (visited on 12/16/2011).

[Xu+09] Tingting Xu, Thomas Pototschnig, Kolja Kühnlenz, and Martin Buss. “A high-speed multi-GPU implementation of bottom-up attention using CUDA”. In: *Robotics and Automation, 2009. ICRA ’09. IEEE International Conference on*. Piscataway: IEEE Press, 2009, pp. 1120–1126. ISBN: 978-1-4244-2788-8. URL: http://portal.acm.org/citation.cfm?id=1703435.1703617 (visited on 12/16/2011).

[Yeh+09] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. “Fool Me Twice: Exploring and Exploiting Error Tolerance in Physics-Based Animation”. In: *ACM Transactions on Graphics* 29.1 (2009), 5:1–11. ISSN: 0730-0301. (Visited on 12/16/2011).

[YPG01] Hector Yee, Sumanta N. Pattanaik, and Donald P. Greenberg. “Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments”. In: *ACM Transactions on Graphics* 20.1 (2001), pp. 39–65. ISSN: 0730-0301. DOI: 10.1145/383745.383748. URL: http://portal.acm.org/citation.cfm?id=383748 (visited on 11/20/2009).

[YQ07] P. Yang and X. Qian. “Direct Computing of Surface Curvatures for Point-Set Surfaces”. In: *Point-based graphics 2007*. Ed. by Baoquan Chen, Matthias Zwicker, Mario Botsch, and Renato Pajarola. Aire-la-Ville and Switzerland: Eurographics Association, 2007, pp. 29–36. ISBN: 978-1-5688-1366-0. DOI: 10.2312/SPBG/SPBG07/029-036. URL: http://diglib.eg.org/EG/DL/WS/SPBG/SPBG07/029-036.pdf (visited on 12/16/2011).

[Zwi+02] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. “EWA Splatting”. In: *IEEE Transactions on Visualization and Computer Graphics* 8.3 (2002), pp. 223–238. ISSN: 1077-2626. URL: http://portal.acm.org/citation.cfm?id=614526 (visited on 11/19/2009).

[Zwi+04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. “Perspective Accurate Splatting”. In: *Proceedings of the 2004 conference on Graphics Interface*. Ontario: Canadian Human-Computer Communications Society, 2004, pp. 247–254. ISBN: 1-56881-227-2. URL: http://dl.acm.org/citation.cfm?id=1006058.1006088 (visited on 12/16/2011).

List of Publications

[SK10] Daniel Schiffner and Detlef Krömker. “Tree-Cut: Dynamic Saliency Based Level of Detail for Point Based Rendering”. In: *Self Integrating Systems for Better Living Environments: First Workshop, Sensyble 2010*. Ed. by Ralf Dörner and Detlef Krömker. Shaker Aachen, 2010, pp. 37–43. URL: `http://www.sensyble.org/lang/de/publikationen/` (visited on 12/16/2011).

[SK11] Daniel Schiffner and Detlef Krömker. “Three Dimensional Saliency Calculation Using Splatting”. In: *Sixth International Conference on Image and Graphics (ICIG), 2011*. Ed. by Nenghai Yu. Piscataway: IEEE, 2011, pp. 835–840. ISBN: 978-1-4577-1560-0. URL: `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6005958` (visited on 12/13/2011).

[SK12a] Daniel Schiffner and Detlef Krömker. “Parallel Treecut-Manipulation for Interactive Level of Detail Selection (to appear)”. In: *WSCG*. 2012,

[SK12b] Daniel Schiffner and Detlef Krömker. “Perception-influenced Animation”. In: *ARCS Workshop on Architectures for Self-Organizing PRivate IT-Spheres*. Springer, 2012, pp. 123–134.

[SSK11] Claudia Stockhausen, Daniel Schiffner, and Detlef Krömker. “Adaptation of User Interfaces based on Physiological Data”. In: *9. Berliner Werkstatt Mensch-Maschine-Systeme (BWMMS)* (2011), pp. 78–79.

List of Figures

List of Tables

List of Listings

Glossary

albedo map The textured, unlit model. This term is defined by deferred shading algorithms.

bidirectional reflectance distribution function Function used for illumination. Extracts lighting information based on given camera and light position using an equation. Is commonly used in computer graphics in the context of bidirectional texturing function[s].

bidirectional texturing function A shader that lookups a bidirectional reflectance distribution function stored in a set of textures. See also shader.

compute shader Universal shader for performing calculations on the gpu. No indirections are required as with old gpgpu methods. Was introduced with shader model 5. See also shader, GPGPU, GPU.

difference of Gaussian Method for extraction of features in saliency calculation. Using different levels, neighboring information is calculated by selecting a lower level of detail.

direct saliency calculation The direct saliency calculation-method uses the saliency splatting method and derives a Layered Conspicuity Map (LCM) for the current scene. This is used in the Dynamic Feedback Method (DFM) to extract the priorities of the scene. The *TreeCut* evaluator strategies use these to adapt the object accordingly..

dynamic bounding volume tree Encodes a hierarchy of bounding volumes. Through its hierarchy, fast updates of inner nodes are possible. Also the updates do not necessarily need to influence inner nodes.

dynamic feedback method The dynamic feedback method combines the necessary calculations and uploads required by the *Feedback System*. These are only necessary when the *Feedback Stage* in the framework has been enabled.

fast Fourier transformation A fast Fourier transformation is the efficient application of a discrete Fourier transformation and its inverse. With the transformation a sequence of signals is decomposed into their different frequencies. For example, image filters have reduced complexity when applied in frequency space.

finite element method Finite element method is used to approximate the solutions of partial different equations. These are applied when detailed simulations of structures are required.

frame A frame contains the resulting color information, which will be used to create an image.

geometry shader The geometry shader is able to generate new primitives during rendering that may also be returned to the rendering subsystem. See also GPU, shader, shader model.

layered conspicuity map A layered conspicuity map combines multiple normalized features. These can be stored within a lookup table and can be utilized by the BSWDF for calculation of saliency values. In a lookup table entry, all depth variants of the layered conspicuity map are referred to as layered conspicuity map set. See also bidirectional texturing function.

layered conspicuity map set All depth variants of a lookup table entry are referred to as a LCM-Set. These are used to allow interpolation between the different levels of the LCM-Set. See also layered conspicuity map, BSWDF, bidirectional texturing function.

level of detail Represents an object using different resolutions by removing primitives in each reduced level. This reduces object's complexity during rendering by providing a reduced and simplified model when applicable. See also primitive.

lookup table saliency calculation In a preprocess step, the lookup table of Layered Conspicuity Map Set (LCM-Set)s is generated. During rendering, the current scene is mapped to the best candidate in the table and a projection matrix is generated. The DFM uploads the stored LCM-Set and projects the current scene onto the pregenerated information. Then the *TreeCut* evaluation strategies adapt the object accordingly..

mass-spring system A mass-spring system consists of a net of springs that are connected with each other by some kind of masses. Such a system can be iteratively processed and the formulas required.

moving least squares Represents a mathematical surface that allows smooth interpolation and extraction of new points.

normal variation Approximates the curvature by averaging the normal variation in a local neighborhood of points.

out-of-core Out-of-core means that the data is not available in local memory but must be loaded from an external source, e.g. the hard disk or a network.

overdraw In the case that multiple pixels are drawn on the same location, usually only the nearest will contribute to the final image. If some type of blending is used, all pixels will be interpolated to acquire the final value. This is used, for example, in the elliptical weighted average approach for point-based rendering. See also point-based rendering, z-buffer, frame.

pixel-space The pixel-space in the mathematical space defined by screen size. Each pixel can be identified by its position on the screen. It is obtained by scaling and translating the normalized device coordinates based on the viewport data.

point-based rendering Direct rendering approach, which does not rely on topological information. This approach is well suited for inner-object level of detail methods as surfel can be exchanged directly. See also surfel,level of detail.

primitive A primitive is a form of representation regarding the provided 3d data. These can be independent or connected, describe a surface or a volume.

rendering subsystem With this term we reference an API for rendering. For example, this could be DirectX^{TM}or openGL..

shader An dynamic allocatable processing unit on the graphics card. It is programmable and directly executed on the GPU. Often a program written using the GPU instruction set is referred to as a shader.

shader model Defines the available instructions and pipeline on a graphics card. Each model introduces either more programmability of the graphics card or provide new stages into the pipeline. See also shader, GPU.

simulation level of detail The equivalent of a level of detail for simulations. Instead of geometric detail, simulation detail is changed.

tessellation shader Complete stage within the shader that enables generation of details in the gpu. No precomputation is required. Consists of the *Hull Shader*,*Tessellator* and *Domain Shader*. See also shader,GPU.

vertex buffer object A vertex buffer object is a buffer reserved on the graphics card memory. It is used for data that needs to be efficiently accessed during rendering, e.g. the vertices of an object.

z-buffer Used during the rasterization process to evaluate, which pixels are actually seen in the final rendering result.

Acronyms

AABB Axis Aligned Bounding Box.

BSWDF Bidirectional Saliency Weight Distribution Function.

CG C for Graphics.

CPU Central Processing Unit.

CSF Contrast Sensitivity Function.

CSS Catmull-Clark Scheme.

EWA Elliptical Weighted Average.

FPS Frames Per Second.

GLSL openGL Shading Language.

GPGPU General Purpose Graphics Processing Unit.

GPU Graphics Processing Unit.

HLSL High Level Shading Language.

HVS Human Visual System.

LGN Lateral Geniculate Nucleus.

MVC Model View Controller.

NDC Normalized Device Coordinates.

NNG Nearest Neighbor Graph.

pixel picture element.

SDK Software Development Kit.

SIMD Single Instruction Multiple Data.

surfel surface element.

texel texture element.

voxel volume element.

WVP World-View-Projection.

A. System Configurations

Test PC 1

Hardware	Type description
CPU	Intel i5 670 with 3.47 GHz
RAM	8.0 GB
OS	Windows 7 64-Bit
GPU	nVidia GeForce 260 GTX

Test PC 2

Hardware	Type description
CPU	Intel Core2 Duo with 3.6 GHz
RAM	2.0 GB
OS	Windows 7 64-Bit
GPU	nVidia GeForce 460 GTX

User Study PC 3

Hardware	Type description
CPU	Intel QuadCore
RAM	4.0 GB
OS	Windows 7 64-Bit
GPU	–
Display	Fujitsu 24” B24W-5 Eco

B. Visual Testing

To validate both the *Feedback System* and the correct functionality of the evaluation strategies, a visual test has been designed. The focus of this test is to determine, whether a spectator is able to distinguish two Level Of Detail (LOD) versions of a shown scene. As visual salience is used in the feedback to identify primitives for removal, higher cognitive processes are excluded from the test.

In the following sections, we will design the test will and present the necessary parts in section 'Preconditions and Test Design'. A series of user tests were performed, and the details regarding the test procedure are given in section 'Test'. The results of all performed tests and the deduced implications for the *Feedback System* are given afterwards.

B.1 Preconditions and Test Design

Our test needs to account for human early vision. We furthermore want to exclude any higher cognitive processes. As no task is employed, a comparison test is chosen.

Participants are asked to select the one of the two images presented, which contains more details or is more visually appealing. The display time of each image pair is equal but varies over all image pairs presented. To avoid that a candidate is able compare the two images explicitly, only one image is presented at a time.

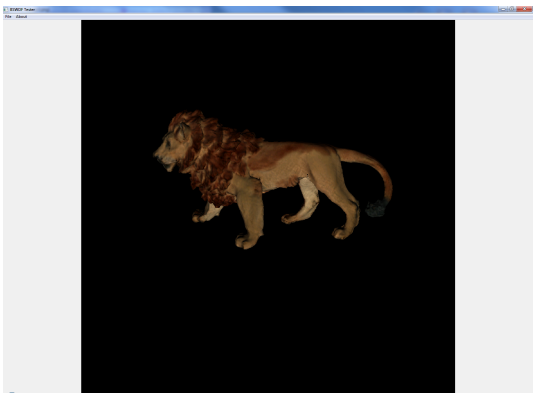
The selection of the image to present first and the display time are random. This allows to test the influence of time in the selection process. No special random number generator has been used neither for generating the order nor for the display times.

As the exchange of the low- and high-detailed images introduces a flicker, a blank screen is presented to the user for a fixed time span. The influence of flicker is reduced while retaining comparison capabilities. As the flicker effect is avoided, decision making is not manipulated. The display duration of the blank screen was set to 0.5 seconds throughout all tests.

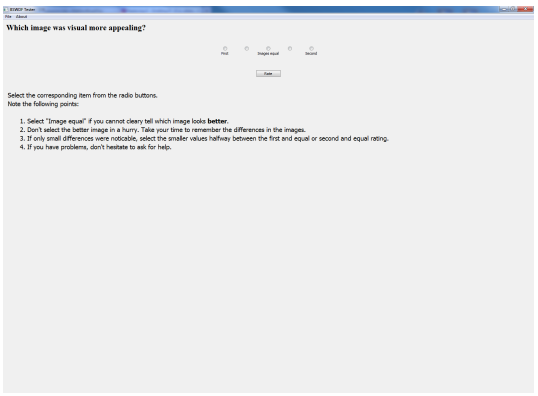
To rate the images a score table with 5 steps is used in the range of [-2,2]. A candidate can choose more fine grained and also rate tendencies. For example, a rated image of -2 means that the first image had the highest visual quality, while a score of 0 indicates that no distinction between the presented images could be made.

B.2 Test

With these preconditions, we have created a test program, which loads a set of pregenerated images. The display time for each image pair is random within a predefined range. In figures B.1a and B.1b screenshots of the generated program can be seen.



(a) The Tester showing a single comparison image.



(b) The score board along with the description of the individual scores. Also some instructions are presented to the participant.

Figure B.1: The created test program, which was used in all performed tests.

Within each image set some images were included that either show very bad results or are equal, i.e. they contain the same content. This allows to extract random picking and non-serious participants. The results of those are excluded for the statistical evaluation. In figure B.2 an example image pair is shown.

In total 30 participants successfully completed the test with various images and display times. Prior to each test, a questionnaire was held to account for possible visual defects of the participant. All of the participants had normal or to normal corrected vision. 2 results were excluded because false ratings were made, which have been detected due to the test images included in each image set.

B.2.1 1st Test

The first test included 15 participants. In each test set, 91 image pairs were shown, which are separated into a normal LOD comparison and reduction technique comparison. The latter image set contained various combinations of the reduction techniques, namely saliency reduction (SAL), point-size reduction (PS) and a combined reduction (SALPS). The display time was set to the range [500,1000] ms. In complete a participant usually required 5 minutes to complete the test.

B.2.2 2nd Test

In the second test, only the LOD comparison is performed to support the findings of the first test. In this test 15 participants were tested, from which 2 were excluded due to invalid selections. 24 image pairs were rated and display times were chosen in the range of [1000,3000] ms. The completion of the test required approximately 2-3 minutes.

Additionally, an image-based compression method was rated against the original version, so that other types of compression are evaluated, too. In our case, a JPEG compression method with very high compression rates was used, ranging from 5%, which produces large visual errors, up to 40%

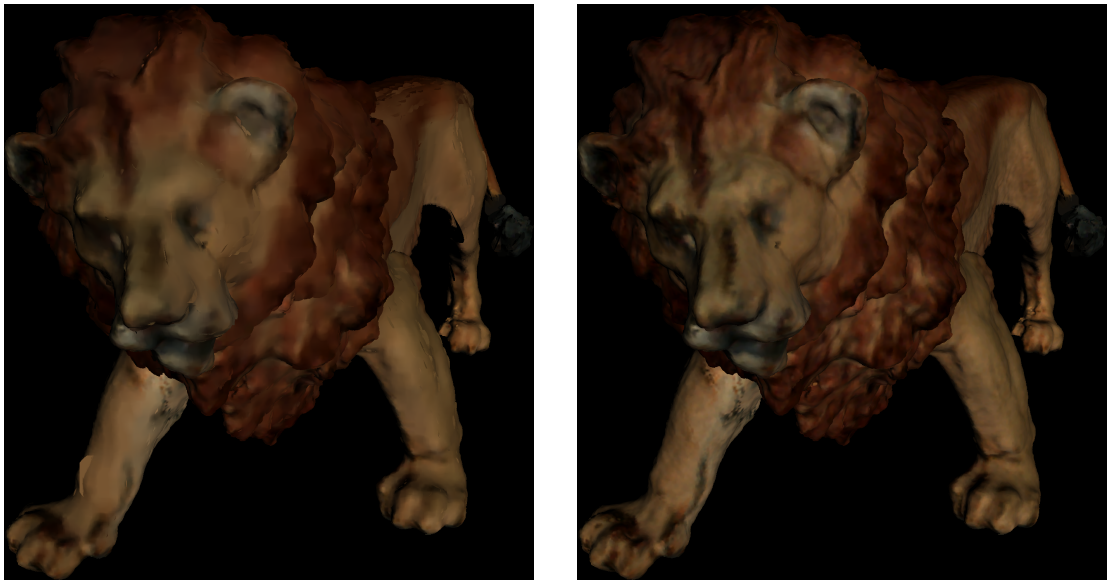


Figure B.2: Example test image pair that was presented during the user test. By test design, a participant had only a short time span to select the more visually appealing image.

of the original source size. In the latter case no differences were perceivable in a pre-test, and thus a rating score of 0 was expected.

B.3 Results

The scores of the test images were removed from the final result set to avoid corruption of the results. As stated before if a participant selected these test images more than twice, the complete ratings of that user are removed.

B.3.1 1st Test

The display times of the first test were evaluated with a linear regression whether a higher score is achieved when a user has more time for viewing an test image. However, in the first test series, the display time did not significantly predict the score ($b = 0.0002$, $t(898) = 1.01$, $p = 0.309 \not\leq 0.005$), and this prediction is also not very good (adjusted $R_2 = 3.824e-05$, $F(1,898) = 1.034$, $p = 0.309$). This may be due to the wrong time ranges selected for testing.

We also extracted the scores for the comparison of LOD techniques. The combined version (SALPS) was rated with a significantly higher score than the other methods (PS and SAL). Both saliency-enhanced LOD techniques received a significantly higher score than the plain PS reduction ($t(179) = -1.689$, $p = 0.04648$). These results were generated using a one sample t-test. A comparison of the SALPS reduction to both PS and SAL methods show a significantly higher rating for the SALPS method ($t(387.418) = -4.7771$, $p = 1.2e-06$). Because multiple tests were made using the same data set, we applied a Bonferroni correction. The significance value for the complete test has

Test	Type	Significant [Yes/No]
Score influenced by time	linear regression	No ($p = 0.309$,$R_2 \approx 0$)
Saliency(SALPS + SAL) better than PS	t-test	Yes ($p = 0.04648$)
SALPS better than SAL	Welch two sample t-test	Yes ($p = 1.2e-06$)

Table B.1: Results of the first visual tests. Type identifies the applied statistical test method. Significant indicates whether the results were significant and includes the statistical values as well. 15 participants conducted this test series.

been set to 0.05 while the sub-tests were subjected to 0.002 for the prediction of the score, 0.047 in case of the comparison of SALPS reduction, and 0.001 in the other case.

The results, which are summarized in table B.1, indicate that the saliency-enhanced LOD techniques increase the visual quality of the presented objects. The application of saliency enhancements results in significantly better results in a forced choice selection. Due to the point-based rendering technique used for generating the test images, any evaluation algorithm using the point-size for deriving the reduction, i.e. in the SALPS and the PS reduction methods, avoids introduction of wrong positioned splats. With other rendering techniques, the plain saliency evaluation is assumed to perform better. Furthermore, we only used static images and we cannot simply extend our findings to a dynamic scenario. However, as long as the dynamic adaptations are processed fast enough, our findings should apply.

B.3.2 2nd Test

In the second test, the highly significant higher rating of the saliency evaluation method has been proven ($t(51) = 3.6124,\ p = 0.0003464 \leq 0.016$). This confirms the results of the first test and endorses the chosen evaluation method.

Despite the fact that a longer display time was chosen, the duration still did not reliably predict the score. We assume that very short times (below 500ms) are necessary to affect the rating. So, we could not prove our assumption that the display time affects the score.

We furthermore evaluated the comparison between the JPEG compressed images and original images. When the JPEG compression factor is below 10% of the original size, the introduced artifacts were too obvious. A highly significant tendency towards the original image has been detected ($t(38) = 5.4371$, $p = 1.6e-06 \leq 0.016$). Interestingly, we found a tendency towards a single version – either the compressed or the original. Each time the images with more detail and less errors were chosen ($t(35) = 0.7766$, $p = 0.4426$). We assume that this is due to the low number of test samples used.

Finally, we evaluated the compression factor that can be achieved with the proposed reduction methods. In this case, only the SALPS reduction method is compared to the PS reduction. The compression in the number of surface elements (surfels) has shown a significantly higher score for

Test	Type	Significant [Yes/No]
SALPS better than PS	t-test	Yes ($p = 0.0003464$)
JPEG compression	t-test	Yes ($p = 1.6e-06$)
SALPS / PS score ratio	linear regression	Yes ($p = 0.000248$,$R_2 = 0.9448$)

Table B.2: Results of the second visual tests. Type identifies the applied statistical test method. Significant indicates whether the results were significant and includes the statistical values as well. 13 participants successfully conducted this test series.

the saliency-enhanced reduction method. This holds as long as the compression is above 60% of the comparison object. There is a linear relation between compression rate and the achieved score ($b = 42.008$, $t(5) = 9.253$, $p = 0.000248 \leq 0.016$). This prediction is also very good as $R_2 = 0.9448$ ($F(1,5) = 85.62,\ p = 0.0002477$). This means, when applying the saliency-enhanced reduction method, the surfel count of an object can be reduced down to 60% without introducing a visible difference. When reducing the number of surfels even more, the artifacts become visible, and the rating towards the saliency-enhanced reduction methods is no longer significant. However, a trend towards the saliency-enhanced method can still be detected. The linear correlation is visualized in figure B.3. All results are summarized in table B.2. In this case, all tests are subjected to a significance of 0.016. This results in an overall significance of 0.05 for the whole test.

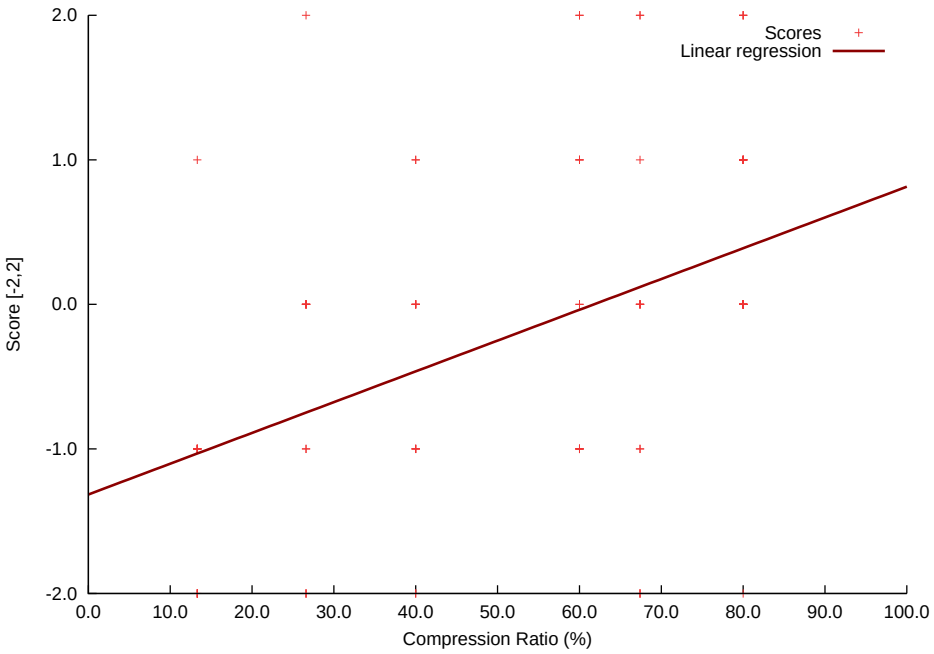


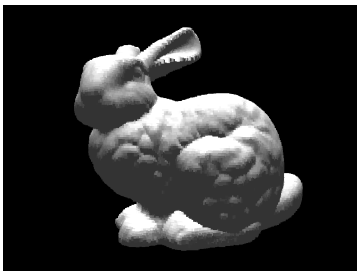
Figure B.3: The linear dependency of the score with respect to the applied compression rate in the *Feedback System*. At 60% of the original object's size, the scores are equal and no distiction could be made by the participants.

C. Results Using the *Feedback System*

In this chapter, we will present some of the visual results achieved using the various methods presented in this thesis. Below each image, the used number of surfels is given.

C.1 Plain *TreeCut* Reductions

The shown images are produced using the priority evaluation strategy on an established *TreeCut*. The comparison is made with an hierarchy based method. In this case, the *TreeCut*'s cut-nodes are in the same level, and the number of surfels is noted below each image.



Original (35286 surfels)

Curvature-based *TreeCut* priority reduction

704 surfels

2186 surfels

9370 surfels

Level-based reduction

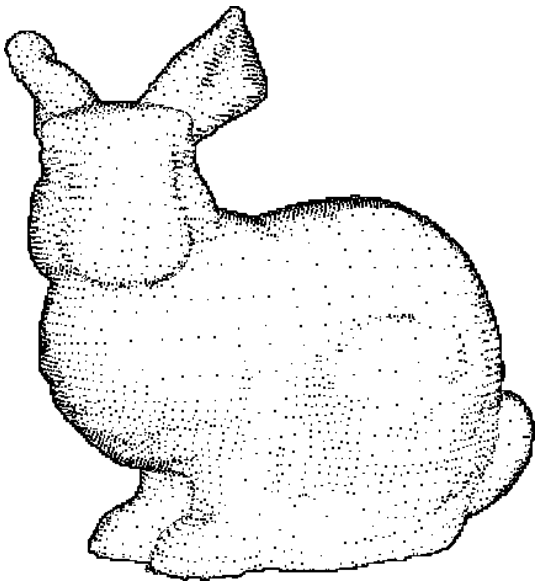
552 surfels

2206 surfels

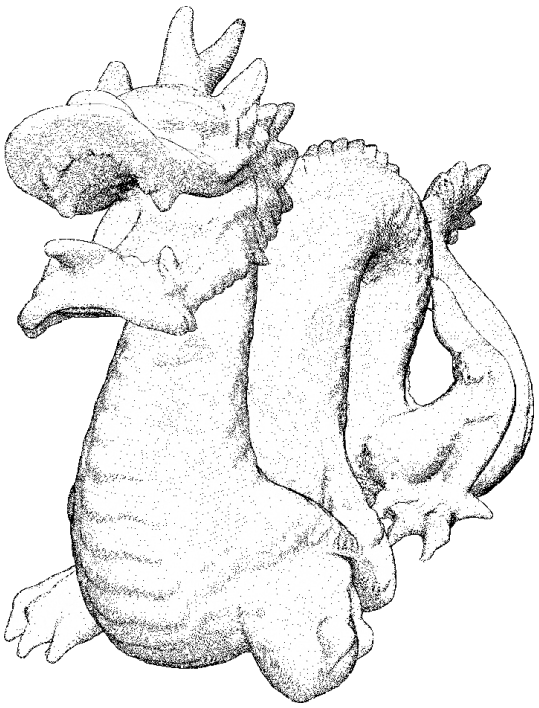
8822 surfels

C.2 Stippling Images

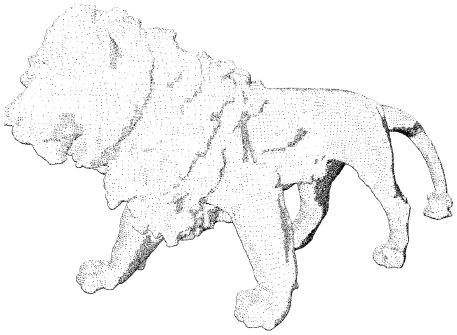
The images contain various objects that were rendered with the stippling approach presented in 'Results Using Bucket-based Cut Evaluation' on page 94. Instead of the used surfel count, the individual parameters are listed below each image.



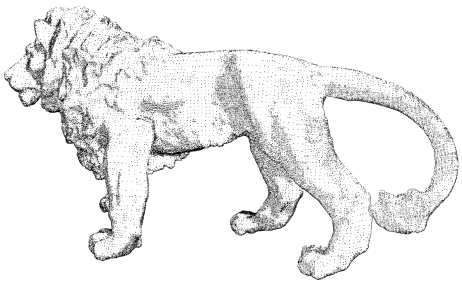
Stanford bunny
MinLvl 5, MaxLvl 7, Gamma 4.8



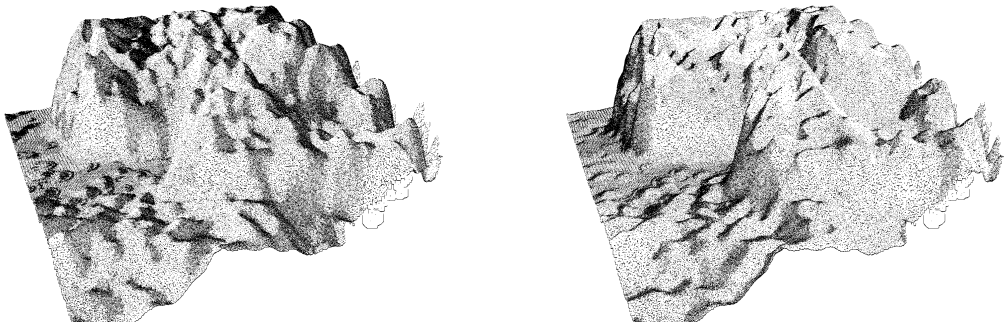
Stanford dragon
MinLvl 5, MaxLvl 10, Gamma 1.0



Stanford lion
MinLvl 5, MaxLvl 13, Gamma 0.3



MinLvl 8, MaxLvl 13, Gamma 3.1

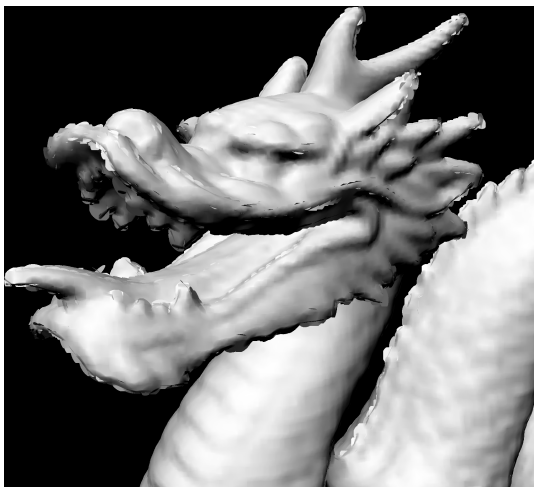


Eochenus tooth generated with the method of Schäfer and Heep [SH10]
Both images have the same settings: MinLvl 5, MaxLvl 12, Gamma 1.3

C.3 Images Used for the Visual Tests

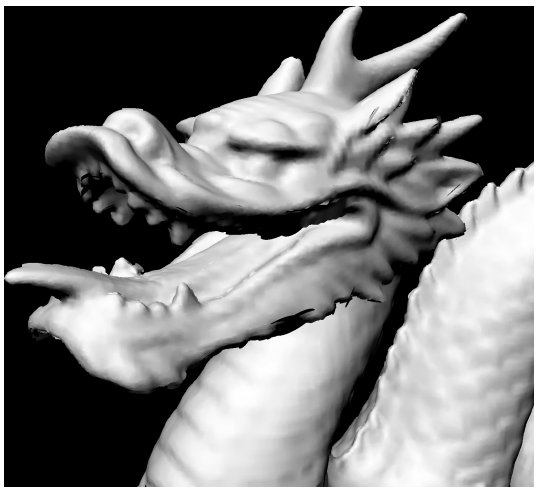
For the rating of the visual quality of our proposed LOD-method, we have generated several test sets. Here, we show two exemplary image pairs that were displayed and rated. For a comparison, a difference image is added here, which was, of course, not shown in the test.

Original (1.279.481 surfels)



Plain point-size reduction

39999 surfels



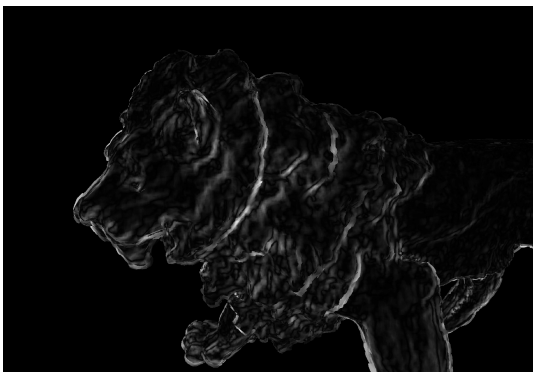
Saliency-enhanced point-size reduction (SALPS)

39998 surfels

Difference of the images

49999 surfels

19999 surfels

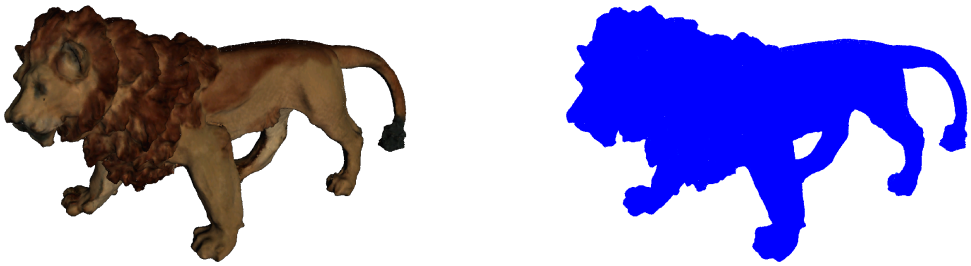


Difference of the images

C.4 LOD-progression Using the *Feedback System*

With our *Feedback System*, continuous LOD adaptations are possible. In the following, we show a progression of the different LOD-stages that were generated when restricting the size of the *TreeCut*. For a comparison, we include absolute difference images to the high quality version. These absolute difference images mark locations where a change has been applied. These, however, do not reflect the magnitude of this change. It is notable that the silhouette of the lion is preserved, even if the number of surfel is reduced to less than 10% of the original count.

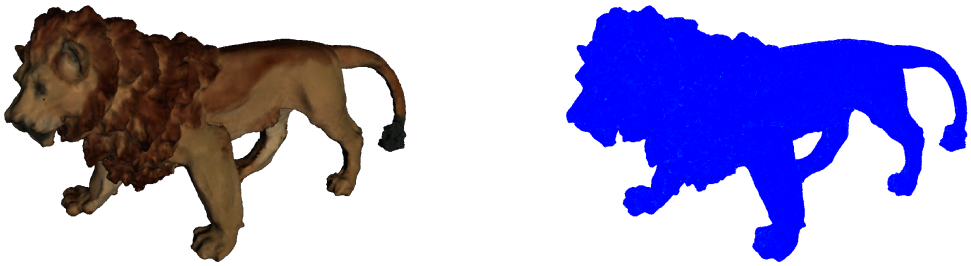
Original (183.408 surfels)



150000 surfels

100000 surfels

80000 surfels



60000 surfels

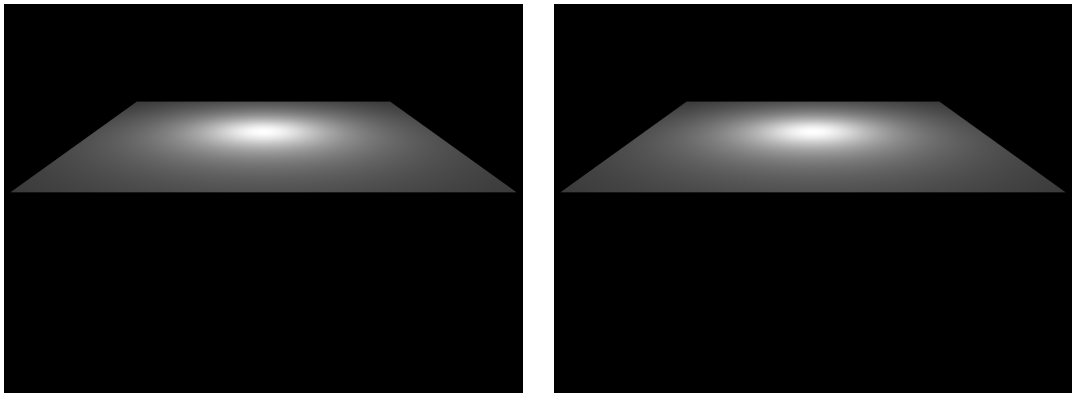
30000 surfels

15000 surfels

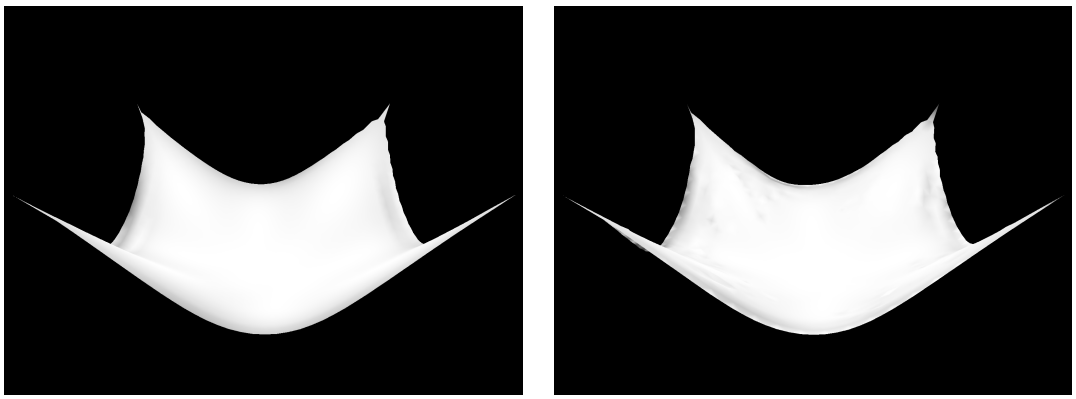
The used color ramp for differences. Blue means no change while red indicates a maximal change of 255 grey scale values. The color images were converted with the NTSC gray scale conversion before the difference has been calculated.

C.5 Perception-influenced Animation

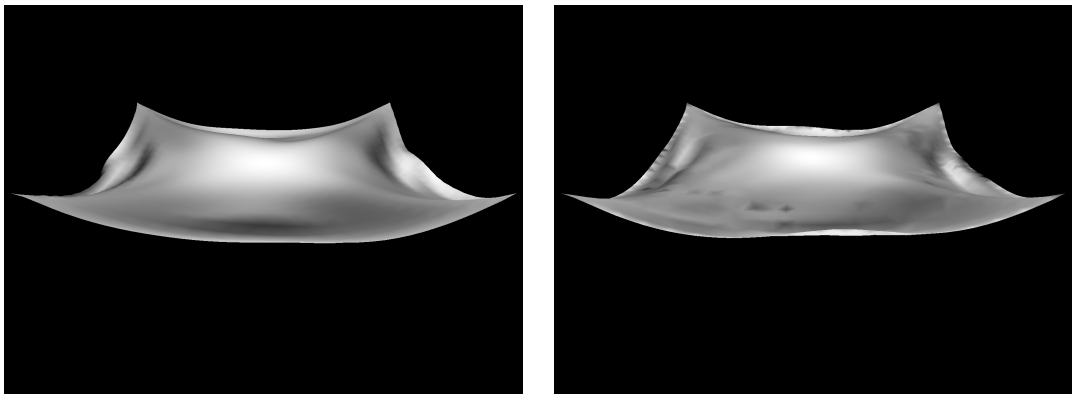
The *Feedback System* is not limited to reduction of static objects, but can be applied to simulations as well. Here, we show some results that we have generated using the *Feedback System*. A soft-body is simulated, and a reduction in size is applied in the progression on the right. In the left, the original soft-body of the `Bullet` physics library is shown. Both objects use 400 simulation nodes, and only gravitational influence is accounted for. A fixed time step is used that is noted below each image pair.



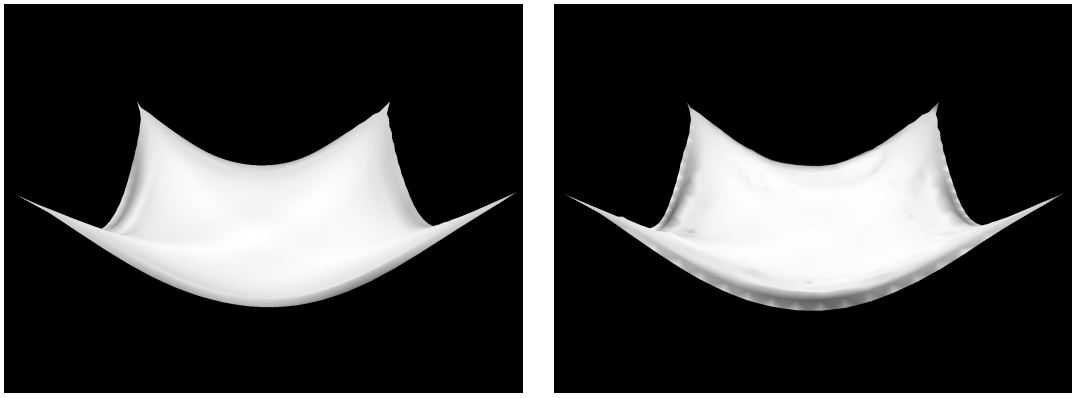
0 steps



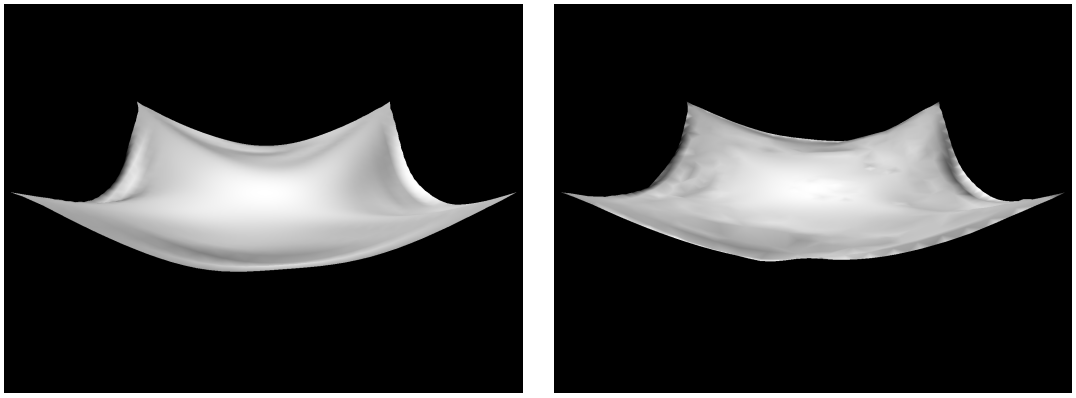
45 steps



90 steps

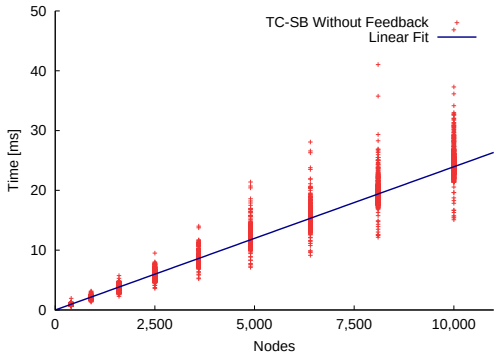


135 steps

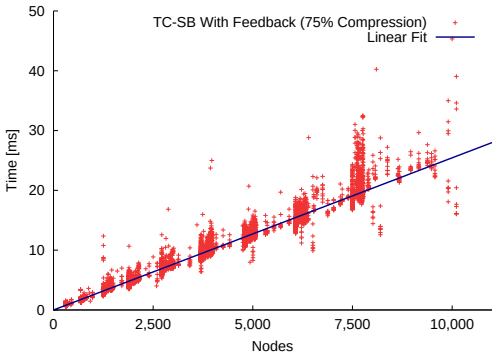


180 steps

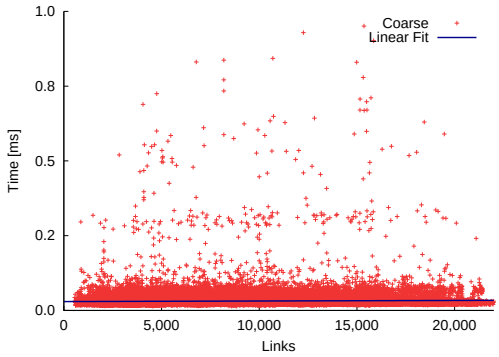
In the next table, the performance results of the *TreeCut-SoftBody* are shown. These were generated using a test set as described in section 'Performance of the Proposed System' on page 165.



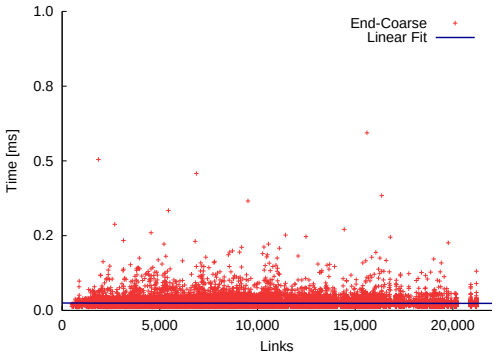
TreeCut-SoftBody without Feedback



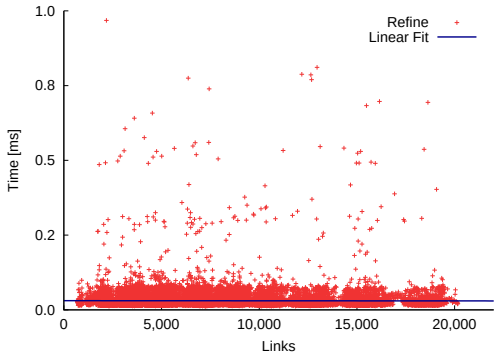
TreeCut-SoftBody withFeedback (reduction to 75%)



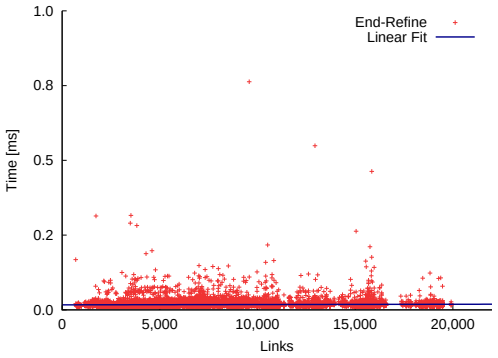
`coarse`-operation



end-`coarse`-operation



`refine`-operation



end-`refine`-operation

D. Curriculum Vitae

Persönliche Informationen

Name	Daniel Schiffner
Geburtsdatum	16.03.1984
Geburtsort	Aschaffenburg
Anschift	Florian-Geyer-Straße 7
	63791 - Karlstein (Main)
E-Mail	dschiffner@gdv.cs.uni-frankfurt.de

Schulische Laufbahn

1990 - 1994	Grundschule Kahl (Main)
1994 - 2004	Spessart Gymnasium Alzenau
	Abschluss Abitur (Leistungsfächer: Mathematik und Physik)
	Gesamtnote: 2.5

Studium

2004 - 2009	Goethe Universität Frankfurt
	Studiengang:
	Diplom Informatik
	Nebenfach:
	Experimentelle Physik
	Vertiefungsfach:
	Computergraphik
	Diplomarbeit:
	Wellenlängenbasiertes Beleuchungsmodell im Shader
	Abschlussnote: 1.0 (mit Auszeichnung)

Persönliche Informationen

2009 - Wissenschaftlicher Mitarbeiter
Professur für Graphische Datenverarbeitung
bei Prof. Dr-Ing. Detlef Krömker

Promotion seit Juni 2009
Mitglied der Grade
Betreuer der Promotion:
Prof. Dr-Ing. Detlef Krömker
Prof. Dr. Ralf Dörner
Prof. Dr. Ulrich Schwanecke

Lehre:

- Betreuung von Vorlesungen / Praktika
  - Computergraphik
  - Animation
  - Visual Computing Praktikum
- Vorlesung “Game Portierung" (SRH Heidelberg)
  In Zusammenarbeit mit Johannes Bufe
- Betreuung von Abschlussarbeiten
  - Jörg Karpf, Diplomarbeit
    “Visuelle Simulation eines Radiosity Algorithm und ihre Anwendung in Lernprozessen”
  - Benjamin Betting, Bachelorarbeit
    “Direct 3D-Output für ein Rendering Framework”
  - Benjamin Bronder & Christian Hornisch, Bachelorarbeit
    “Interactive Gorilla”
  - Sven Pallus, Diplomarbeit
    “Point-Based Animation”
  - Tobias Berger, Bachelorarbeit
    “Dynamische Tree-Cut Unterteilung bei Animation auf Punkt-basierten Modellen”